(12)特許協力条約に基づいて公開された国際出願

(19) 世界知的所有権機関
国際事務局

(43) 国際公開日
2004 年4 月29 日 (29.04.2004)

PCT

(10) 国際公開番号
WO 2004/036463 A1

(51) 国際特許分類[7]: G06F 17/50

(21) 国際出願番号: PCT/JP2003/012839

(22) 国際出願日: 2003 年10 月7 日 (07.10.2003)

(25) 国際出願の言語: 日本語

(26) 国際公開の言語: 日本語

(30) 優先権データ:
特願 2002-300073
2002 年10 月15 日 (15.10.2002) JP

(71) 出願人 (米国を除く全ての指定国について): 株式会社ルネサステクノロジ (RENESAS TECHNOLOGY CORP.) [JP/JP]; 〒100-6334 東京都 千代田区 丸の内二丁目４番１号 Tokyo (JP).

(72) 発明者; および

(75) 発明者/出願人 (米国についてのみ): 谷本 匡亮 (TANIMOTO,Tadaaki) [JP/JP]; 〒100-6334 東京都 千代田区 丸の内二丁目４番１号 株式会社ルネサステクノロジ内 Tokyo (JP). 鎌田 丈良夫 (KAMADA,Masurao) [JP/JP]; 〒100-6334 東京都 千代田区 丸の内二丁目４番１号 株式会社ルネサステクノロジ内 Tokyo (JP).

(74) 代理人: 玉村 静世 (TAMAMURA,Shizuyo); 〒101-0052 東京都 千代田区 神田小川町２丁目１０番地 新山城ビル４２号 Tokyo (JP).

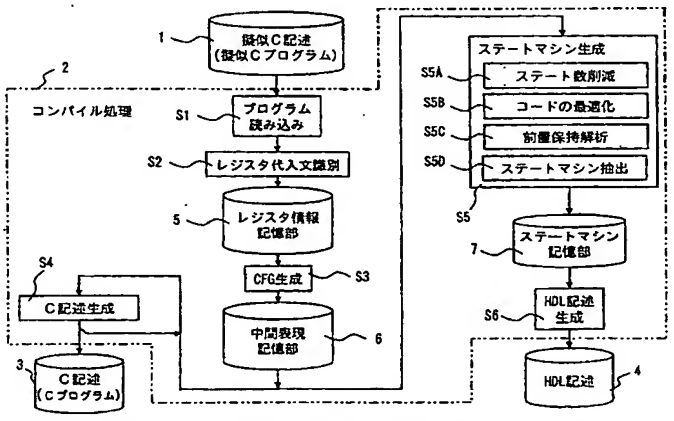
(81) 指定国 (国内): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) 指定国 (広域): ARIPO 特許 (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), ユーラシア特許 (AM,

[続葉有]

(54) Title: COMPILER AND LOGIC CIRCUIT DESIGN METHOD

(54) 発明の名称: コンパイラ及び論理回路の設計方法



1...PSEUDO C DESCRIPTION (PSEUDO C PROGRAM)
2...COMPILER PROCESSING
S1...READ PROGRAM
S2...IDENTIFY REGISTER ASSIGNMENT STATEMENT
5...REGISTER INFORMATION STORAGE SECTION
S3...CFG GENERATION
S4...C DESCRIPTION GENERATION
3...C DESCRIPTION (C PROGRAM)
6...INTERMEDIATE EXPRESSION STORAGE SECTION
S5...STATE MACHINE GENERATION
S5A...REDUCTION OF NUMBER OF STATES
S5B...OPTIMIZATION OF CODE
S5C...FRONT HOLDING ANALYSIS
S5D...STATE MACHINE EXTRACTION
7...STATE MACHINE STORAGE SECTION
S6...HDL DESCRIPTION GENERATION
4...HDL DESCRIPTION

(57) Abstract: A compiler is supplied with a pseudo C description (1) which can describe parallel operation at the statement level by a clock boundary and a register assignment statement with a cycle accuracy, identifies the register assignment statement (S2), generates an executable C description (3), extracts a state machine in which the number of states has been reduced, and judges whether any loop executed by 0 cycle is present (S5). If none, the compiler generates a circuit description (4) capable of synthesizing a logic. Thus, a pseudo C description having C description in which a clock boundary is explicitly inserted is input. Since the pseudo C description capable of parallel description at the statement level by the register assignment statement is input, it is possible to express the pipeline operation accompanied by stall operation.

[続葉有]

WO 2004/036463 A1

AZ, BY, KG, KZ, MD, RU, TJ, TM), ヨーロッパ特許 (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, RO, SE, SI, SK, TR), OAPI 特許 (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

添付公開書類:
— 国際調査報告書

２文字コード及び他の略語については、定期発行される各*PCT*ガゼットの巻頭に掲載されている「コードと略語のガイダンスノート」を参照。

(57) 要約:

コンパイラは、クロック境界及びレジスタ代入文によりステートメントレベルでの並列動作の記述をサイクル精度で記述可能な擬似Ｃ記述（１）を入力とし、レジスタ代入文の識別を行い（Ｓ２）、実行可能なＣ記述（３）を生成すると共に、状態数削減を行ったステートマシンを抽出し、０サイクルで実行されるループが存在するか否かを判定し（Ｓ５）、もしなければ、論理合成可能な回路記述（４）を生成する。これにより、クロック境界を明示的にＣ記述内に挿入した擬似Ｃ記述を入力し、レジスタ代入文によるステートメントレベルでの並列記述を可能にした擬似Ｃ記述を入力するから、ストール動作を伴うパイプライン動作が表現可能である。

明　細　書

コンパイラ及び論理回路の設計方法

技術分野

本発明はプログラム記述からシミュレーション用のプログラム記述又はハードウェアを特定する回路記述を自動生成する技術に関し、例えばパイプライン動作される論理回路、例えばＣＰＵ（Central Processing Unit）等の論理回路の設計に適用して有効な技術に関する。

背景技術

プログラム言語を用いてディジタル回路の回路記述を生成する技術がある。特許文献１に記載の技術では、レジスタを示す変数と、レジスタの入力を示す変数とに分け、モジュール部での処理の後に第２の変数から第１の変数に一括して代入する一括代入部を設けている。特許文献２には、汎用プログラム言語で回路動作を記述したプログラムの中から、順次制御する部分を特定処理部で特定し、その後、変換処理部で、前記順次制御する部分の記述を、ステートマシンとして動作するように汎用プログラム言語を用いて変換し、その変換後のプログラムを取得し、続いて、プログラム生成処理部で、前記変換後のプログラムの中から並行動作する部分を抽出し、この抽出部分の全てをアクセスするプログラムを生成する、というものである。

特許文献１：特開２００２－４９６５２号公報

特許文献２：特開平１０－１４９３８２号公報

発明の開示

特許文献１によれば、①回路動作を示すモジュール、②レジスタ代入を行う一括代入部、③クロック同期で繰り返すループ部の３つの構成からなっており、特に③内で①の実行後に②を実行する事を特徴としている。しかしながら、①がクロック境界を含まず、必ず③に含まれる構成となるため、複数サイクルにまたがる回路動作を記述する為には、回路動作をクロック境界で分割する必要がある。例えば、ある条件が成立したときは、前サイクルで実行した回路動作の途中から回路動作を行うという記述を行わなければならないが、そのような記述を行うのは困難である。特に、ストール動作を伴うパイプライン動作を行う回路を特許文献１に示す方法で記述すると、煩雑な作業を伴い、かつプログラム記述が複雑なものになる虞のあることが本発明者によって見出された。

特許文献２によれば、①汎用言語で記述したプログラムから順次処理部を識別し、ステートマシンを表す汎用プログラム記述に変換、②関数レベルでの並列性の抽出、③ハード化するプログラムとそれを制御するソフトプログラムの結合の自動化、④順次処理部内でハード化する際にフリップフロップやラッチを必要とする部分を識別してＨＤＬに変換、の４つを特徴点がある。しかしながら、クロック境界を明示的に与えられる手段がなく、サイクル精度での記述を直接行う事が出来ない。特許文献２の実施例によれば、クロック境界は関数から関数への間であり、例えばある条件が成立したときは、前サイクルで実行した回路動作の途中から回路動作を行うという記述を行うのが困難である。特に、ストール動作を伴うパイプライン動作を行う回路を特許文献２に示す方法で記述することは可能であるが、煩雑な作業を伴い、かつプログラム記述が複雑になる虞のあることが本発明者によって見出された。

本発明の目的は、　本発明の目的は、クロック境界を明示的に記述したプログラム記述からハードウェア記述を自動生成することができる

コンパイラを提供することにある。

本発明の別の目的は、ストール動作を伴うパイプライン動作が可能な回路のプログラム記述又は回路記述を容易に得る事ができるコンパイラを提供することにある。

本発明の更に別の目的は、ストール動作を伴うパイプライン動作が可能な回路の設計を行うことができる論理回路の設計方法を提供することにある。

本発明の上記並びにその他の目的と新規な特徴は本明細書の以下の記述と添付図面から明らかにされるであろう。

本願において開示される発明のうち代表的なものの概要を簡単に説明すれば下記の通りである。

〔１〕本発明の概要を全体的に説明する。即ち、クロック境界（記述子＄）及びレジスタ代入文（演算子＝＄を挟む記述）によりステートメントレベルでの並列動作の記述をサイクル精度で記述可能な擬似Ｃ記述（１）を入力とし、レジスタ代入文の識別を行い（Ｓ２）、実行可能なＣ記述（３）を生成するする（Ｓ３およびＳ４）と共に、状態数削減を行ったステートマシンを抽出し、０サイクルで実行されるループが存在するか否かを判定し（Ｓ５）、もしなければ、論理合成可能な回路記述（４）を生成する（Ｓ６）。

上記より、クロック境界を明示的にＣ記述内に挿入した擬似Ｃ記述を入力し、レジスタ代入文によるステートメントレベルでの並列記述を可能にした擬似Ｃ記述を入力するから、ストール動作を伴うパイプライン動作が表現可能である。

擬似Ｃ記述から一般のＣコンパイラによるコンパイルが可能なＣ記述を出力することができる。状態（ステート）数削減を行うので、記述で与えたクロック境界の数＋１以下のステート数のステートマシンを

伴う回路記述を出力することができる。

ステートマシンを意識する事なくプログラム・レベルで機能設計を行う事ができるため、記述量が低減され、開発期間の短縮のみならず品質向上にも寄与する。

また、一般のクロック境界を指定しないプログラム・レベルでの記述では表現できない、バス・インターフェース回路や調停回路の記述が可能となる。特に、レジスタ代入が記述可能である為、ステートメントレベルでの並列性を考慮した記述を行う事が可能であり、ストール動作を伴うパイプライン動作のような複雑な回路動作をＣ記述よりも少ないコード量で容易に記述可能である。

また、一般のＣコンパイラでコンパイル可能なＣ記述へ変換する為、高速なシミュレーションが可能となり、機能検証工数の大幅な低減が可能となる。従って、機能設計における論理設計、論理検証の双方の大幅な工数削減が可能となる。

クロック境界を指定したプログラム記述からミーリー（Ｍｅａｌｙ）型のステートマシンが生成可能であるので、プログラム・レベルでのモデル検査を行う事が可能である。

高位合成ツールが不得意とする、サイクル精度を要求される例えば、キャッシュ・コントローラやＤＭＡコントローラの開発に適用可能であり、設計期間の短縮に大きく寄与する。

〔２〕本発明に係るコンパイラの第１形態では、コンパイラは、所定のプログラム言語を流用して記述された第１プログラム記述（１）を回路記述（４）に変換可能であって、前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文(演算子＝＄を挟む記述）とクロック境界記述（＄）を含み、前記回路記述は、前記第１プログラム記述が特定する回路動作を実現するハードウェアを所定のハ

ードウェア記述言語で特定する。

本発明に係るコンパイラの第２形態では、コンパイラは、所定のプログラム言語を流用して記述された第１プログラム記述を所定のプログラム言語を用いた第２プログラム記述（３）に変換可能であり、前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文（演算子＝＄を挟む記述）とクロック境界記述（＄）を含む。前記第２プログラム記述は、前のサイクルの状態を参照可能に前記レジスタ代入文を変形した変形代入文（１３）と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文（１２）とを含む。

本発明に係るコンパイラの第３形態では、コンパイラは、所定のプログラム言語を流用して記述された第１プログラム記述（１）を、所定のプログラム言語を用いた第２プログラム記述（３）と回路記述（４）に変換可能である。前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含む。前記第２プログラム記述は、前のサイクルの状態を参照可能に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述とを含む。前記回路記述は、前記第２プログラム記述で定義されるハードウェアを所定のハードウェア記述言語で特定する。

前記所定のプログラム言語は例えばＣ言語である。前記ハードウェア記述言語は例えばＲＴＬレベルの記述言語である。

〔３〕本発明に係る論理回路の設計方法の第１形態では、タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文（演算子＝＄を挟む記述）とクロック境界記述（＄）を含む第１プ

ログラム記述（１）を入力する第１処理（Ｓ１）と、前記第１プログラム記述に基づいて前記タイミング仕様を満足する回路情報を生成する第２処理と、を含む。

前記第２処理は、前記第１プログラム記述を変換して、前記レジスタ代入文が入力変数と出力変数を用いて変形される（Ｓ２）と共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する（Ｓ４）記述（１３，１２）を含む第２プログラム記述（３）を、前記回路情報として生成する処理を含んでよい。

前記第２処理は、前記第２プログラム記述に基づいて前記タイミング仕様を満足するハードウェアを所定のハードウェア記述言語で特定するための回路記述（４）を更に別の前記回路情報として生成する処理を含んでよい。

前記第２プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含んでもよい。

上記第２処理に関し、前記レジスタ代入文が入力変数と出力変数を用いて変形される（Ｓ２）記述（１３）を含む第２プログラム記述（５）と、前記クロック境界記述に対応させて前記入力変数を出力変数に代入する（Ｓ４）記述（１２）を含む第３プログラム記述（３）とを、分けて把握することも可能である。このとき、第３処理によりシミュレーションは第３プログラム記述に基づいて行うことになる。

〔４〕本発明に係る論理回路の設計方法の第２形態では、タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含む第１プログラム記述を入力する入力処理（Ｓ１）と、前記レジスタ代入文が入力変数と出力変数を用いて変形される（Ｓ２）と共に前記クロック境界記述に対応させて前記入力変数

を出力変数に代入する（Ｓ４）記述（１３，１２）を含み、前記所定のプログラム言語で記述された第２プログラム記述を生成する変換処理とを含む。

前記変換処理は、第１プログラム記述に基づいてＣＦＧを生成する過程で、前記ＣＦＧに前記クロック境界記述に対応してクロック境界ノードを設定し、前記クロック境界ノードの後に、前記レジスタ代入記述を挿入する処理であってよい。

第２プログラム記述に対してそのＣＦＧを利用しながらステート遷移毎の変数表を作成しながらコード最適化を行う最適化処理を更に含んでもよい。

前記変数表においてステート間で変数に変化のない部分を前置保持を要する部分として抽出し、抽出された部分に、出力変数に入力変数を代入する代入記述を追加する前置保持処理を更に含んでもよい。

前記前置保持処理を経た変数表の各ステート遷移毎の変数と引数に基づいてステートマシンを構成するコードの抽出を行う抽出処理を更に含んでもよい。

前記抽出処理で抽出されたステートマシン構成コードと第２プログラム記述を参照しながら、前記回路仕様を満足する回路のハードウェアを所定のハードウェア記述言語で記述する回路記述を生成する処理を更に含んでもよい。

前記第１プログラム記述に対して０サイクルで実行されるループが存在するか否かが判定され、存在しないと判別されたときに前記変換処理が行なわれる。

図面の簡単な説明

第１図は本発明に係る論理回路の設計方法を例示するフローチャー

トである。

第２図は第１図の設計方法を適用して設計すべき回路例を示すブロック図である。

第３図は第２図の回路動作仕様を示すタイミングチャートである。

第４図は第２図の設計対象回路の擬似Ｃプログラムを例示する説明図である。

第５図はレジスタ代入文識別処理（Ｓ２）によって得られる追加変数宣言の記述とレジスタ代入文書き換えの記述を示す説明図である。

第６図は擬似Ｃ記述に基づくＣＦＧ作成過程の一つの過程を示す説明図である。

第７図は擬似Ｃ記述に基づくＣＦＧ作成過程の別の過程を示す説明図である。

第８図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第９図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１０図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１１図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１２図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１３図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１４図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１５図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１６図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１７図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１８図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第１９図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第２０図は擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

第２１図は擬似Ｃ記述に基づくＣＦＧ作成過程の最終過程を示す説明図である。

第２２図は説明を簡単化するために第２１図のＣＦＧに対してクロック境界や分岐の始点・終点、及びループの始点・終点の情報を付加していないＣＦＧを例示する説明図である。

第２３図は第２２図のＣＦＧに対するフラグ挿入状態を例示する説明図である。

第２４図はレジスタ代入記述挿入文の挿入位置をＣＦＧ上で例示する説明図である。

第２５図はＣ記述生成処理（Ｓ４）を経て得られる実行可能な変換Ｃ記述（Ｃプログラム）の最初の一部を例示する説明図である。

第２６図は第２５図に続く実行可能な変換Ｃ記述（Ｃプログラム）の一部を例示する説明図である。

第２７図は第２６図に続く実行可能な変換Ｃ記述（Ｃプログラム）の

最後の部分を例示する説明図である。

第２８図はステート数削減処理の第１のルールを示す説明図である。

第２９図はステート数削減処理の第２のルールを示す説明図である。

第３０図は第２２図のＣＦＧに対してステート数削減を行った結果を例示する説明図である。

第３１図はステート数削減等の処理を行ったＣＦＧに対してステートの割り当てを行った状態を例示する説明図である。

第３２図は前記コード最適化の処理を説明するために特別に簡素化した例としてコード最適化対象とされる擬似Ｃプログラムを示す説明図である。

第３３図は第３２図の擬似Ｃプログラムに基づいて得られたＣＦＧを例示する説明図である。

第３４図は第３３図のＣＦＧに対してステート割り当てが行なわれた状態を例示する説明図である。

第３５図は第３４図のＣＦＧに対してステートマシン生成のための変数表作成処理過程の最初の状態を例示する説明図である。

第３６図は第３５図に続く変数表作成処理過程の次の状態を例示する説明図である。

第３７図は第３６図に続く変数表作成処理過程の次の状態を例示する説明図である。

第３８図は第３７図に続く変数表作成処理過程の次の状態を例示する説明図である。

第３９図は第３８図に続く変数表作成処理過程の次の状態を例示する説明図である。

第４０図は第３９図の生成過程を経て生成された変数表を例示する説明図である。

第４１図は第４０図の変数表に対して冗長ステートメント削除を行ったとき、削除されるべきステートメントを例示する説明図である。

第４２図は第４１図に対して冗長ステートメントが削除された結果の変数表を例示する説明図である。

第４３図は冗長ステートメントが削除された結果をＣＦＧで示す説明図である。

第４４図は第４２図の変数表に対してローカル変数削除を行ったとき削除されるべき変数を例示する説明図である。

第４５図はローカル変数削除処理が行なわれた結果をＣＦＧで示す説明図である。

第４６図は冗長ステートメント削除処理及びローカル変数削除処理が行なわれて最終的に更新された変数表を例示する説明図である。

第４７図は後工程の前置保持解析により変数表に前置保持“retain“の記述が追加された状態を例示する説明図である。

第４８図はコードの最適化として更に演算式の簡約化を行った例をＣＦＧで示す説明図である。

第４９図は第３２図乃至第４８図で特別に簡素化した別の例を用いて説明したコード最適化の処理を第３１図に示されるステート割り当てが行われた後に施すことによって得られる最適化後のＣＦＧを例示する説明図である。

第５０図は第４９図に対する最適化処理後の変数表を示す説明図である。

第５１図は前置保持解析のアルゴリズムを例示する説明図である。

第５２図は前置保持解析の結果に対応する変数表を示す説明図である。

第５３図は第５２図に対し“retain”を実際のコードで上書きした変

数表を示す説明図である。

第５４図は開始ステートＳＴ０におけるステートマシン抽出処理を示す説明図である。

第５５図は第５４図に対しretain情報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子を示す説明図である。

第５６図は開始ステートＳＴ１におけるステートマシン抽出処理を示す説明図である。

第５７図は第５６図に対しretain情報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子を示す説明図である。

第５８図は開始ステートＳＴ２におけるステートマシン抽出処理を示す説明図である。

第５９図は第５８図に対しretain情報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子を示す説明図である。

第６０図はＨＤＬ記述生成処理（Ｓ６）にて生成されたＨＤＬ記述の最初の一部を示す説明図である。

第６１図は第６０図に続くＨＤＬ記述の一部を示す説明図である。

第６２図は第６１図に続くＨＤＬ記述の最後の部分を示す説明図である。

発明を実施するための最良の形態

《設計方法の概略》

第１図には本発明に係る論理回路の設計方法が例示される。同図に示される設計方法は、擬似Ｃ記述（擬似Ｃプログラム）１の作成、擬似Ｃプログラム１に対するコンパイル処理２に大別される。コンパイル処理２では、擬似Ｃプログラム１を、レジスタ代入記述を変形代入文とした擬似Ｃプログラム（５に格納）、および実行可能なＣ記述（Ｃプログラ

ム）３に変換し、また、そのＣプログラム３をＲＴＬ(Register Transfer Level）などのＨＤＬ（Hardware Description Language）記述４に変換する。

前記擬似Ｃプログラム１は、サイクル精度で回路動作を特定可能とするクロック境界記述（単にクロック境界とも記す）及びレジスタ代入文を含み、ステートメントレベルでの並列記述を可能にしたプログラムである。擬似Ｃ記述とは、前記クロック境界及びレジスタ代入文が定義されていないない所謂ネイティブのＣ言語記述とは相違するという意味で用いられている。プログラム言語としてＣ言語以外の高級言語をベースとすることを妨げるものではない。

コンパイル処理２は、図示を省略するコンピュータ装置がコンパイラを実行し、擬似Ｃプログラム１を読み込んで行なわれる。先ず擬似Ｃプログラム１が読み込まれる（Ｓ１）。読み込まれた擬似Ｃプログラム１に対しては、レジスタ代入文の識別が行なわれ、識別されたレジスタ代入文を、前のサイクルの状態を参照可能に変形し、換言すれば、入力変数と出力変数を用いて変形する（Ｓ２）。変形されたレジスタ代入文を変形代入文とも称する。レジスタ代入文が変形代入文に変形された擬似Ｃプログラムはレジスタ情報記憶部５に格納される。レジスタ代入文が変形代入文に変形された擬似Ｃプログラムは前記レジスタ情報記憶部５から取り出されて、そのコントロール・フロー・グラフ（以下ＣＦＧと記す）が生成される（Ｓ３）。生成されたＣＦＧは中間表現記憶部６に格納される。前記中間表現記憶部６に格納されたＣＦＧ及び前記レジスタ情報記憶部５に格納された擬似Ｃプログラムは、実行可能なＣ記述プログラムに変換される（Ｓ４）。例えば、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文が挿入される。換言すれば、クロッ

ク境界記述に対応させて前記変形代入文の入力変数を出力変数に代入するレジスタ代入記述挿入文が挿入される。

前記擬似Ｃプログラム５等に基づいてＨＤＬ記述４を得る場合、先ずそれらを入力してステートマシンの生成が行なわれる（Ｓ５）。ステートマシン生成（Ｓ５）は、ステート数削減処理（Ｓ５Ａ）、コードの最適化（Ｓ５Ｂ）、ＨＤＬ記述に則するための前置保持解析（Ｓ５Ｃ）、及びステートマシン抽出（Ｓ５Ｄ）に大別される。ステート数削減処理（Ｓ５Ａ）とコードの最適化（Ｓ５Ｂ）は最適化処理の範疇に属する処理と把握してもよい。コードの最適化（Ｓ５Ｂ）の段階では、０サイクルで実行されるループが存在するか否かを判定し、もしなければ、ＨＤＬ記述に則するための前置保持解析（Ｓ５Ｃ）、及びステートマシン抽出（Ｓ５Ｄ）が行われる。前記Ｃ記述プログラムを得るときには、例えばクロック境界ノードに前記レジスタ代入記述挿入文を挿入すればよかったが、ＨＤＬ記述を得るときはクロック境界でレジスタ値が変化しない場合にもそれを明示的に記述しておくことが必要とされる。そのために、前置保持解析（Ｓ５Ｃ）が行なわれる。生成されたステートマシンはステート遷移毎の変数表に基づいて生成される。生成されたステートマシンはステートマシン記憶部７に保持される。保持されたステートマシン等に基づいてＨＤＬ記述４が生成される（Ｓ６）。

ＨＤＬ記述４は論理合成ツールを利用することによって論理回路図データに変換可能にされる。前記Ｃ記述３は前記論理合成される論理回路のシミュレーションなどに利用される。

以下に、上記擬似Ｃプログラムとそのコンパイル処理を詳細に説明する。以下の詳細説明は第２図の回路に第３図の仕様を満足させる回路の設計を一例とする。

《設計対象回路》

第２図には第１図の設計方法を適用して設計すべき回路例が示される。設計対象回路１０はストール動作を伴うパイプライン加算回路である。その動作仕様は以下の通りである。

（１）入力信号 valid_a が立ち上がると、信号レベルのハイレベルとなったサイクルの入力信号 a の値を取り込む。ここでは valid_a が立ち上がり変化を問題にする。

（２）入力信号 valid_a の立ち上がりの次サイクル以降で、入力信号 valid_b の信号レベルがハイレベルとなると、そのサイクルでの入力信号 b の値を取り込む。入力信号 valid_b に対してはレベル検出だけで充分とされ、エッジ変化の検出は不要とされる。

（３）上記（１）（２）の動作で a と b が取り込まれたなら、その次サイクルで a と b の加算結果を出力信号 out により送出し、その同一サイクルに出力信号 valid_out の信号レベルをハイレベルとし、次サイクルで出力信号 valid_out の信号レベルをロウレベルとする。

（４）出力信号 out は（１）（２）（３）の動作での新たな加算結果が代入されない限り、同じ値を出力する。

（５）出力信号 valid_out は（１）（２）（３）の動作で出力信号 out へ新たな加算結果が代入されたサイクルのみ信号レベルがハイとなり、それ以外はロウレベルを出力する。

第３図には第２図の回路動作仕様を示すタイミングチャートである。同図において、出力データ送出と入力データ取り込みが同一サイクルで行われており、パイプライン動作となっている。例えば a2 の入力と a1+b1 の出力が並列化されている。また、入力信号 valid_a の立ち上がりの次サイクル以降で入力信号 valid_b の値が１となった次のサイクルで出力データ送出が行われる為、ストール動作を伴うパイプライン動作となっている。例えば b1 の取込み後における b2 の取込みは２サイク

ル待たされている。

《擬似Ｃプログラム》

第４図には前記設計対象回路１０の擬似Ｃプログラムが例示される。第４図に記述において１１は、設計対象回路１０の回路動作を記述した回路動作記述部である。同図に示される擬似Ｃプログラムの記述は以下の通りである。即ち、

１行目：Ｃ言語でのライブラリ呼び出し、

２～７行目：関数 pipeline のプロトタイプ宣言部、

８～１４行目： main 関数部、

９～１０行目： main 関数のローカル変数宣言部。出力信号はポインタ型で宣言、

１１～１２行目：main 関数のローカル変数の初期化（出力信号のみ初期化、特に出力信号に対してＲＴＬへの変換時にレジスタが推定される場合ここで指定した初期値がリセット値となる）、

１５～３６行目：pipeline 関数部、

１８～２０行目：pipeline 関数のローカル変数宣言部（特にローカル変数に対してＲＴＬへの変換時にレジスタが推定される場合ここで指定した初期値がリセット値となる）、

２１～３５行目：回路動作記述部１１、である。

回路動作記述部１１の詳細は以下の通りである。即ち、

２１、３５行目：無限ループにより回路を表現、

２２行目:入力変数 valid_a のローカル変数 valid_a_tmp へのレジスタ代入文（ここで、0x0001&valid_a により、入力変数 valid_a の有効ビット幅が１ビットである事を指定している）、

２３行目：valid_a が 1'b1 で valid_a_tmp が 1'b0 であるか否かの判定文（即ち、valid_a が立ち上がりであるか否かの判定文。特に、

0x0001&valid_a_tmp により、ローカル変数 valid_a_tmp の有効ビット幅が１ビットである事を指定している）、

２４行目:入力信号 a のローカル変数 a_tmp への代入文(特に、0x7FFF&a により、入力変数 a の有効ビット幅が１５ビットである事を指定している）、

２５行目：クロック境界、

２６行目：goto ラベル、

２７～２８行目:入力変数 valid_b が 1'b1 であれば、ローカル変数 b_tmp に入力変数 b を代入し、そうでなければクロック境界を１つまたいでラベルＬへ分岐する事を表している（特に、0x0001&valid_b により、入力変数 b の有効ビット幅が１ビットである事を、0x7FFF&b により、入力変数 b の有効ビット幅が１５ビットである事を表している）、

２９行目：ローカル変数 a_tmp とローカル変数 b_tmp の和の出力変数 out へのレジスタ代入文、

３０行目：定数 0x0001 の出力変数 valid_out へのレジスタ代入文、

３１行目:２３行目の if 文の判定が成立しなかった場合の分岐。即ち、valid_a が立ち上がりでなかった場合の分岐を表す、

３２行目：クロック境界、

３３行目：定数 0x0000 の出力信号 valid_out へのレジスタ代入文、である。

上記記号“＄”はクロック境界記述を意味し、記号“＝＄”レジスタ代入を意味する。それらはＣ言語の汎用的な記述子及び演算子ではない。これを用いた擬似Ｃプログラムは、その意味においてＣ言語を流用したプログラム記述と言うことができる。

上記回路動作記述部１１より明らかなように、クロック境界記述及びレジスタ代入文によりステートメントレベルで並列動作をサイクル精

度で簡単に記述可能になる。サイクル精度とは、クロックサイクルとの同期が意図される、ということである。

第４図の回路動作記述部１１の記述内容について説明する。入力変数valid_a をローカル変数 valid_a_tmp に代入する事で、if 文による valid_a の立ち上がり判定を行い、もし立ち上がりであった場合は、ローカル変数 a_tmp に入力信号 a を取り込み、次のサイクルで入力信号 valid_b が 1'b1 であるか否かを判定する。もしそうなら入力信号 b の値をローカル変数 b_tmp に代入し、そうでなければ次のサイクルでもう一度入力信号 valid_b が 1'b1 であるか否かを判定する。これを入力信号 valid_b が 1'b1 となるまで繰り返す。この動作がストール動作に対応している。さて、ローカル変数 a_tmp と b_tmp の和は取り込んだ a と b の値の和を表しており、それを出力変数 out へレジスタ代入し、同時に 1'b1 を出力信号 valid_out へレジスタ代入している。これにより、入力信号 a と b を取り込んだ１サイクル後での加算結果と valid_out 信号が 1'b1 である事を表現している。if 文による valid_a の立ち上がり判定を行い、立ち上がりでない場合は、１サイクル後に 1'b0 を valid_out へレジスタ代入している。valid_a の立ち上がりは高々２サイクルに１回しか起こり得ないので、変数 out への新たな代入が２９行目で行われた時のみ valid_out が 1'b1 となり、それ以外の場合は、1'b0 となる。

第４図の第２２行におけるレジスタ代入文は、サイクル精度で動作を特定するのに順序回路としてのレジスタを想定しており、左辺（valid_a_tmp）はレジスタの出力、即ち前サイクルの値を保持している変数として把握可能である。レジスタ代入文の右辺（0x0001&valid_a）は現時点のレジスタ入力として把握可能である。また、第４図の第２９行目及び第３０行目に記載のレジスタ代入文に関しては、その後の第３２行におけるクロック境界記述でクロックが消費さ

れるようになっているが、第２図及び第３図の回路仕様ではその次サイクルで out を出力するとあり、結果として、out、valid_out に関しては必然的にサイクル精度の記述が必要になるため、それらの記述にはレジスタ代入文が用いられている。

《レジスタ代入文識別》

次にレジスタ代入文識別処理Ｓ２について説明する。前記レジスタ代入文識別処理部では、代入文であって、＝と右辺の間に＄が付加された文を識別し、回路動作記述部１１内のレジスタ代入文、レジスタ代入文の左辺の変数の型と初期値を記憶し、識別したレジスタ代入文

signal_latched = $ signal;を

signal_latched_i = signal;

signal_latched = signal_latched_o;

の記述に変更する。signal_latched_i は現時点の入力が与えられる入力変数、signal_latched_o は 1 サイクル前の出力が当てられる出力変数として把握することが可能である。変数宣言部に変更により生じた新たな変数

signal_latched_i, signal_latched_o

を先に記憶しておいた変数の型と初期値を参照して追加する。例えば、

unsigned char signal_latched = 0x01;

の場合は、

unsigned char signal_latched_o = 0x01;

unsigned char signal_latched_i;

を追加する。特に、レジスタ代入の左辺の変数が、ポインタ型の場合（記号*が付されている）は、そのポインタ型を用いて変数宣言を行う。例えば、

unsigned char *signal_latched;

の場合は、

unsigned char signal_latched_o = 0x01;

unsigned char signal_latched_i;

を追加する。特に、追加対象となった変数に対して、同じ型で初期値を０としたフラグ変数も追加予定として、記憶する。この例の場合、

unsigned char flg_signal_latched = 0x00;

を追加予定変数として記憶する。尚、上記変更を行った記述も記憶する。また、変数の初期値はＨＤＬ変換時に該変数へのレジスタ推定が行われた場合、リセット時の値として用いる。

第５図にはレジスタ代入文識別処理Ｓ２によって得られる結果が例示される。第４図の擬似Ｃプログラム対して追加変数宣言の記述と変形代入文（レジスタ代入文書き換え）１３の記述が変更されている。

《ＣＦＧ生成》

次にＣＦＧ生成処理について説明する。ＣＦＧとは、一般に各関数内部において制御の流れを示すグラフを意味する。

ＣＦＧ生成処理では、回路動作記述部１１を読み込んで、ＣＦＧの作成を行う。特に while や for 等のループ及び if や case 等の条件分岐、goto 文によるラベルへのラベル分岐を識別する為のノードを持つＣＦＧの作成を行う。要するに、while や for 等のループ及び if や case 等の条件分岐、goto 文によるラベルへのラベル分岐をノードに持つＣＦＧを作成する。各文をプログラムの終了迄読み込み、以下の手順１）～７）でノードを作成しながらプログラムの流れに沿って、ノード間の接続を有向辺（向きが付いている辺）で接続する事でＣＦＧを作成する。第６図から第２１図には手順１）～７）によるＣＦＧの作成過程が順を追って示される。各図にはループ文スタック、分岐文スタック、生成途中のＣＦＧが示される。

1）ループの開始であれば、ループ文スタックにその行番号とwhileやforなどのループを表す終端記号を登録し、ループ開始ノード（ＮＤｓ）を作成し、行番号と終端記号をノードに付加する。また、forやwhileループ終了条件があれば、その条件を適当な記号に代入し、出力枝に付加し、付加した条件を割り当てた記号との対で記憶する。

2）ループの終了であれば、ループ文スタックから先頭にある情報を取り去り、ループ終了を表すループ終了ノードを作成し、行番号と“end of 終端記号”をノードに付加する。但し、continueやbreakはループの終了としては扱わない。また、do-whileループ終了条件があれば、その条件を適当な記号に代入し、出力枝に付加し、付加した条件を割り当てた記号との対で記憶する。

3）条件分岐の開始であれば、分岐文スタックにその行番号とifやcase等の分岐を表す終端記号を登録し、条件分岐開始ノードを作成し、行番号と終端記号をノードに付加する。また、分岐条件を適当な記号に代入し、出力枝に付加し、付加した条件を割り当てた記号との対で記憶する。

4）条件分岐の終了であれば、分岐文スタックから先頭にある情報を取り去り、条件分岐終了を表す条件分岐終了ノードを作成し、行番号と“end of 終端記号”をノードに付加する。

5）ラベルであれば、ラベルを表すラベルノードを作成し、行番号とラベル記号をノードに付加する。

6）クロック境界であれば、クロック境界ノードを作成し、行番号と＄をノードに付加する。

7）上記以外であれば、行番号と文を付加したノードを作成し、1）～6）の何れかに出会うまでノードをマージする。

上記手順によるＣＦＧが作成されるが、以下の説明では、その説明を

簡単化するために、第２２図に例示されるように、クロック境界や分岐の始点・終点、及びループの始点・終点の情報を付加していないＣＦＧを用いて説明を行う。特に、クロック境界ノードのみ黒丸で、それ以外のループ、条件分岐、ラベル分岐ノードを白丸で表現する。

《Ｃ記述生成》

前記Ｃ記述生成処理Ｓ４について説明する。Ｃ記述生成処理Ｓ４では、前記レジスタ代入文識別処理で追加予定変数として記憶しておいた変数で、レジスタ代入部識別処理で変更した部分（変形代入文）の直下に対応するフラグ変数に１を代入する文の挿入を行い、レジスタ代入文の左辺の変数への代入文でレジスタ代入文でない代入文の直下に対応するフラグ変数に０を代入する文の挿入を行う。また同時に、ローカル変数宣言部に、レジスタ代入文識別部で記憶しておいた変数宣言を追加する。第２３図では flg_valid_a_tmp=1、flg_valid_out=1 のフラグが挿入されている。

次にレジスタ代入記述挿入文が決定される。前記レジスタ代入文識別処理Ｓ２において、識別されたレジスタ代入文の右辺の変数全てに対して、レジスタ代入記述挿入文が作成される。即ち、

レジスタ代入文：

```
signal_latched = $ signal;
```

変更後の記述：

```
signal_lacthed_i = signal;
signal_latched = signal_latched_o;
```

追加された変数：

```
signal_latched_i, signal_latched_o, flg_signal_latched
```

とされている場合、下記記述

```
signal_latched_o = signal_latched_i;
```

```
if (flg_signal_latched==1) signal_latched = signal_latched_o;
```

を作成する。これをレジスタ代入文識別処理で識別したレジスタ代入文の右辺の変数全てに対して作成する。例の場合には、下記記述

```
valid_a_tmp_o = valid_a_tmp_i;
if (flg_valid_a_tmp==1) valid_a_tmp = valid_a_tmp_o;
out_o = out_i;
if (flg_out==1) *out = out_o;
valid_out_o = valid_out_i;
if (flg_valid_out==1) *valid_out = valid_out_o;
```

が得られる。

上記レジスタ代入記述挿入文は、第２４図に例示されるように、クロック境界ノードの直下に挿入される。第２４図においてレジスタ代入記述挿入文には参照符号１２が付されている。このようにして行なわれるＣ記述への変換は、各ノードに付加された行番号等の情報を元に、深さ優先探索等のアルゴリズム（ＤＦＳ）を用いて、ＣＦＧを探索する事で挿入文の順番を考慮して行えば良い。尚、適度にコメント文を挿入しても良い。

第２５図乃至第２７図には上記Ｃ記述生成処理Ｓ４を経て得られる実行可能な変換Ｃ記述（Ｃプログラム）３の全体が例示される。

《ステートマシン生成－ステート数削減》

前記ステートマシンの生成処理Ｓ５について説明する。ステート数削減処理Ｓ５Ａは例えば第１又は第２のルールに従って行なわれる。ステート数削減処理の第１のルールは第２８図に例示される。即ち、ループ開始・終了ノード、条件分岐開始・終了ノード、ラベル分岐ノードの何れかであって、入力辺が複数あるノードを探索し、その入力辺の内２つ以上の入力辺にクロック境界がある場合は、同図に示すグラフ変形を行

う。ステート数削減処理の第２のルールは第２９図に例示される。即ち、ループ開始・終了ノード、条件分岐開始・終了ノード、ラベル分岐ノードの何れかであって、出力辺が複数あり且つ出力辺に付加された条件が入力信号も出力信号の何れも含まず、２本以上の出力辺にクロック境界が付加されたノードを探索し、その前段のクロック境界が出力辺のクロック境界を含まない場合、同図に示すグラフ変形を行う。第３０図には第２２図のＣＦＧに対してステート数削減を行った結果が例示される。

《ステートマシン生成－コード最適化》

コード最適化処理Ｓ５Ｂでは前記ステート数削減等の処理を行ったＣＦＧに対しては第３１図に例示されるようにステートの割り当てを行う。第３１図に従えば、

回路動作部の開始文に対応するＣＦＧ上のノードに初期ステートを割り当て、ＣＦＧ上のクロック境界ノードにステートを割り当てる。但し、開始ノードへの入力辺が１つしか存在せずクロック境界が付加されている場合は、既に割り当てた初期ステートを削除する。尚、最適化の第１ルールにより、初期ステート削除が起こる必要十分条件は、開始ノードへの入力辺が１つしか存在せずクロック境界が付加されている事である事に注意することが望ましい。また、得られるステート数は、必ず回路動作部に記述したクロック境界の数＋１以下となる事に注意すべきである。

ここで、前記コード最適化の処理を、特別に簡素化した別の例を用いて、第３２図乃至第４８図を参照しながら説明する。

第３２図はコード最適化対象とされる擬似Ｃプログラムを示す。この擬似Ｃプログラムに基づいて得られたＣＦＧは第３３図に例示される。第３４図には第３３図のＣＦＧに対してステート割り当てが行なわれた状態を例示する。

第３５図から第４０図まではステートマシン生成のための変数表作成処理の様子が順を追って例示される。変数表の作成は、以下の（１）～（３）の手順で行う。（１）ローカル変数を取得し、（２）関数の引数を取得し、（３）割り当てたステートからステートに到達するまでＣＦＧを下位側に辿って、ステート遷移を識別すると共に変数の定義・参照の情報を取得する。この段階で、両端がクロック境界ではないループが発見されると、ゼロサイクル・ループを検出したとして、ユーザに通知し、処理を終了。ゼロサイクル・ループの発生は、生成される回路に組合せ回路からなるループ回路が存在する事を意味しており、ループ回路の存在は生成される回路に重大なミスがあることを意味する。第３５図にはステートＳＴ０からＳＴ１への一つのステート遷移におけるローカル変数と引数が例示される。第３６図にはステートＳＴ０からＳＴ１への別のステート遷移におけるローカル変数と引数が例示される。第３７図にはステートＳＴ０からＳＴ２へのステート遷移におけるローカル変数と引数が例示される。第３８図にはステートＳＴ１からＳＴ０へのステート遷移におけるローカル変数と引数が例示される。第３９図にはステートＳＴ２からＳＴ０へのステート遷移におけるローカル変数と引数が例示される。第３５図から第３９図に示される夫々のステート遷移で得られたローカル変数と引数に基づいて、第４０図に例示される変数表が生成される。第４０図の変数表の記述において、def[n]：n行目で変数定義されている事を表し、

use@var[m]：m行目で変数 var への代入に用いられている事を表し、

pred(cond){...}：条件 cond の分岐が成立した場合、{...}が実施される事を表し、

def[l]use：１行目で自変数への代入に用いられている事を表し、

use@pred(cond)：条件 cond で用いられている事を表す、とされる。

最適化処理は例えば第４０図の変数表に基づいて行なわれる。最適化処理の一つは冗長ステートメントの削除である。

冗長ステートメントの削除として、第１に、同一変数に対して、ステート遷移のカラム内で def が２つ以上存在する場合には、１）又は２）の処理を行う。即ち、

１）下記１－１），１－２）を def の後段に存在する pred(cond){...}の手前まで(pred(cond){...}の有無に関わらず)実施する。１－１）：def の後段に use を伴う def がない場合は、最後の def に対応するステートメントのみ残す。１－２）：def の後段に use を伴う def がある場合は、use を伴う def の後段に use を伴わない def があれば、その def のみを残し、そうでなければ use を伴う def の前段の def と use を伴う def を残し、これを変化が無くなるまで繰り返し、残った def に対応するステートメントのみ残す。

２）def の後段に pred(cond){...}が無ければ終了し、あれば下記２－１），２－２）を実施する。２－１）：pred(cond){...}の条件が def の結果を参照している場合には終了とする。２－２）:pred(cond){...}の条件が def の結果を参照していない場合は、１）へ分岐とする。

冗長ステートメントの削除処理として、第２に、use がどのステート遷移にも存在しない変数は削除とする。

上記処理手順により第４０図の変数表に対して冗長ステートメント削除を行ったとき、削除されるべきステートメントは第４１図に示される。同図において削除されるべきステートメントには斜め破線が明示されている。第４２図には冗長ステートメントが削除された結果の変数表が例示される。第４３図には冗長ステートメントが削除された結果をＣＦＧで表している。

最適化処理のもう一つはローカル変数の削除である。このローカル変

数の削除処理として、第１に、各変数のステート遷移カラムに於いて、下記１）～３）を左から順次変化が無くなる迄実施する。即ち、

１）def の後段に pred(cond){...}を挟まず use が存在する場合には、１－１）、１－２）、１－３）、１－４）を行う。１－１）：use 自体が use@pred の場合は代入操作を実施し、def を削除し、１－２）の場合は@の変数がローカル変数で use@pred として用いられている場合は代入操作を実施せず、１－３）：@の変数がローカル変数で use@pred として用いられていない場合は代入操作を実施し、def を削除し、１－４）：@の変数が引数の場合は代入操作を実施し、def を削除する。

２）def の後段に pred(cond){...}を挟んで use が存在し、pred(cond){...}の条件で用いられている変数が引数の場合は１－１）から１－４）を適用する。

３）def の後段に pred(cond){...}を挟んで use が存在し、pred(cond){...}の条件で用いられている変数がローカル変数の場合には、３－１）、３－２）を行う。３－１）：pred(cond){...}の条件が def の結果を参照していない場合は１－１）から１－４）を適用し、３－２）：pred(cond){...}の条件が def の結果を参照している場合は代入操作を実施しない。

ローカル変数の削除処理として、第２に、def がどのステート遷移カラムにも存在しない変数は削除し、代入操作後のＣＦＧを再び解析して、変数表を更新する。

第４２図の変数表に対してローカル変数削除を行ったとき、削除されるべき変数は第４４図に示される。同図において削除されるべき変数には斜め破線が明示されている。第４５図にはローカル変数削除処理が行なわれた結果をＣＦＧで表している。

第４６図には冗長ステートメント削除処理及びローカル変数削除処

理が行なわれて最終的に更新された変数表が例示される。この変数表により、必要となるローカル変数が管理されることになる。第４６図において、def が存在しない部分では、出力変数とローカル変数の前置保持が必要である事が識別できる。ステートの遷移において当然前置保持されなければならないからである。従って、その部分には、前置保持を必要とすることが容易に識別可能になる。第４７図に例示されるように、後工程の前置保持解析により、その部分に、前置保持“retain“が追加されることになる。

コードの最適化として更に、第４８図に例示されるような演算式の簡約化が行なわれる。

《ステートマシン生成－前置保持解析》

ここからの説明は再度第２図及び第３図の仕様を満足させる回路設計の例に話しを戻す。第３２図乃至第４８図では特別に簡素化した別の例を用いて前記コード最適化の処理を説明したが、第３１図に示されるステート割り当てが行われた後に、それと同様の最適化処理を施すことにより、第４９図の最適化後のＣＦＧを得ることができ、また、第５０図の変数表が得られる。最適化処理後の第５０図の変数表には前置保持“retain”は明示されていない。次に説明する前置保持解析で取得される。

第５１図には前置保持解析のアルゴリズムが例示される。前置保持解析は、出力変数とローカル変数に対して、状態遷移のカラムにて def が全く存在しない場合、その状態遷移では前置保持が必要となる。また、def が存在したとしても、pred()が付加されている場合は、各出力変数・ローカル変数の各状態遷移に対して、第５１図に示されるような図を作成して、pred()による分岐の中でどの部分で前置保持が必要となるかを識別する。特に、レジスタ代入文識別処理で新たに追加したローカ

ル変数に対しては、_i が付加されている変数のみに対して前置保持が必要かの解析を行う。また、例え、変数表に pred()の情報がなくても必要ならＣＦＧを解析し直して付加する。

第５１図に示される図の作成は、pred()の条件を分岐として、use、def 等のノードを持つ木を作成する事で行う。そして木の def より下位の部分木を削除し、最上位ノード以外で下位ノードに def が存在しないノードを前置保持が必要なノードとして識別する。

ここで、最上位ノードとは、木の根から一番距離が近い def か use のノードまでのノードとその兄弟ノード全てを指す。

例えば、変数 var の状態遷移 STn－＞STm での変数表からの情報が、pred(cond_0){pred(cond_1){use@var_1[j], pred(cond_2){def[k], pred(cond_3){def[s]}}}である場合、第５１図のようになる。

第５２図には前置保持解析の結果に対応する変数表が例示される。前置保持を要する部分には“retain”が追加される。

変数表において“retain”の部分に追加すべき実際のコードは変数表から取得することができる。即ち、前置保持解析結果の変数表からの情報取得処理では、変数表のカラムで retain が挿入された、出力変数・ローカル変数を取得し、例えば変数名が

１）レジスタ代入文の左辺の変数の場合は、sig = sig_o;

２）レジスタ代入文識別部で追加した変数であって、_i が付加されている変数の場合は、sig_i = sig_o;

３）その他の変数の場合は、nxt_sig = sig;

として記憶しておく。特に、retain に pred()が付加されている場合は、例えば、pred(cond_0){pred(cond_1){pred(!cond_2){retain}}}に対しては、変数が３）の場合で変数名が sig の場合、pred(cond_0){pred(cond_1){pred(!cond_2){nxt_sig = sig}}}として記

憶する。以上の情報を変数表に上書き登録する。第５３図には“retain”が実際のコードで上書きされた変数表が例示される。更に、nxt_sigといった具合に nxt_を付加した変数を記憶しておく。第５３図の例の場合、nxt_を付加した変数は、a_tmpのみである。

《ステートマシン生成－ステートマシン抽出》

次にステートマシン抽出処理Ｓ５Ｄについて説明する。ステートマシン抽出処理Ｓ５Ｄでは、割り当てた各ステートから深さ優先探索でクロック境界即ちステートであって初期ステートでないステートに到達するまで探索し、その探索で得られたループでも条件分岐でもラベル分岐でもないノードの情報を取得し、変数表の retain 情報とマージして、ＨＤＬ記述に用いるステートマシンの抽出を行う。例えば第５４図には開始ステートＳＴ０の例が示される。ステートの記述は、特に制限されないが、各ステートからＤＦＳでコードを生成する。この場合ステート変数は、nxt_state = ST0;等の形式として、コード生成を行う。

特に、retain 情報にて nxt_が付加された変数はもとの変数名ではなく nxt_が付加された変数名を用いてＨＤＬ記述に用いるステートマシンの抽出を行う。また、信号と定数との＆演算はビット幅解析に用いたので、不要となるため削除する。尚、定数は入力左辺のビット数を勘案してＨＤＬの２進表記に変換してＨＤＬ記述に則した記述とする。

変数表の retain 情報の取得では、各状態遷移カラムから、深さ優先探索を開始したステートと同じステートを開始ステートするカラムを全て取得し、retain が開始ステートのみに依存するか、到達ステートにも依存するか、または到達ステートと分岐条件に依存するかを識別し、開始ステートにのみ依存する場合以外は、retain 情報の pred()とＣＦＧの分岐条件を比較する事で、ＨＤＬコードの適切な位置に retain 情報として変数表に格納した代入式を挿入する。第５５図には retain 情

報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子が例示される。

第５４図及び第５５図には開始ステートＳＴ０におけるＨＤＬ記述に則したステートマシン記述の取得例が示される。第５６図及び第５７図の例は開始ステートＳＴ１におけるＨＤＬ記述に則したステートマシン記述の取得例が示される。第５８図及び第５９図の例は開始ステートＳＴ２におけるＨＤＬ記述に則したステートマシン記述の取得例が示される。

《ＨＤＬ記述生成処理》

ＨＤＬ記述生成処理Ｓ６において、モジュール宣言は、回路動作記述部を含むＣ記述の関数宣言から、型とポインタを表す＊を削除したものに clk と reset_n を加えたものをＨＤＬ記述として生成する。入出力宣言は、前記関数宣言での引数であって、代入式の左辺にのみ存在する変数を出力とし、代入式の右辺にのみ存在する変数を入力とし、ビット幅はＣ記述の記述内容で説明した方法で識別し、ＨＤＬ記述として生成する。reg 宣言はＣ記述に記載されていたローカル変数で、これまでの変換仮定で最終的に残った変数と、これまでの変換仮定で追加された変数とを識別し、clk と reset_n の reg 宣言文とともにＨＤＬ記述として生成する。ＣＦＧ生成過程で分岐条件に割り当てた変数の wire 宣言のＨＤＬ記述を生成し、前記割り当てた変数への分岐条件の代入文を assign 文としてＨＤＬ記述を生成する。また、割り当てたステートを２進数で表す為の parameter 宣言文のＨＤＬ記述を生成する。

また、レジスタ代入文に関しては、全てのレジスタ代入文とその右辺の変数宣言を取得し、例えば、取得した情報が

```
unsigned char sig1_latched = 0x00;
unsigned char sig2_latched = 0x00;
```

```
unsigned short out;
sig1_latched = $ sig1&0x03;
sig2_latched = $ sig2&0x13;
out          = $ exe_result&0x1FFF;
```

の場合、

```
always @ (posedge clk or negedge reset_n) begin
  if (!reset_n) begin
    sig1_latched_o <= 2'b00;
    sig2_latched_o <= 3'b000;
  end
  else begin
    sig1_latched_o <= sig1_latched_i;
    sig2_latched_o <= sig2_latched_i;
    out_o          <= out_i;
  end
end
```

のようなＨＤＬ記述を生成する。

次いで、ステートマシン抽出部で得た nxt_が付加された変数の記憶を参照し、その変数の宣言部を取得し、例えば、

この例の場合、a_tmp が対象となるが、

```
unsigned short nxt_a_tmp = 0x0000;
```

であり、reg 宣言記述生成時に、

```
a_tmp = $ 0x7FFF&a;
```

なる代入から有効ビット幅が１５ビットである事が解っているので、下記

```
always @(posedge clk or reset_n) begin
```

```
if (!reset_n) begin
  state = ST0;
  a_tmp = 15'b000000000000000;
end
else begin
  state = nxt_state;
  a_tmp = nxt_a_tmp;
end
end
```

の記述を生成する。

また、抽出されたステートマシンのＨＤＬ記述をつなげ、各ステートでの代入文の左辺に対して、レジスタ代入の左辺の変数とレジスタ代入文識別部で追加した変数には、対応する_o の変数を代入し、それ以外の変数には初期値を代入した文を作成し、nxt_state=ST0;なる文を作成し、case 文の default に対応する部分を作成し、それもつなげ、右辺の変数と wire 宣言した変数を or で並べ、下記

```
always @ (state or c1 or c2 or valid_a_tmp_i or valid_a_tmp_o or
valid_a_tmp or a_tmp or
        valid_out_i or valid_out_o or out_i or out_o) begin
  case(state[1:0])
  endcase
end
```

の記述を生成し、case 文の間につなげたＨＤＬ記述を挿入し、最後の行に endmodule を付加する事でＨＤＬ記述を生成する。行数は付加しただけである。

第６０図乃至第６２図にはＨＤＬ記述生成処理Ｓ６にて生成された

ＨＤＬ記述が例示される。

以上説明した設計方法によれば、以下の作用効果を得る。

クロック境界を明示的にＣ記述内に挿入した擬似Ｃ記述を入力し、レジスタ代入文によるステートメントレベルでの並列記述を可能にした擬似Ｃ記述を入力するから、ストール動作を伴うパイプライン動作が表現可能である。

ステートマシンを意識する事なくプログラム・レベルで機能設計を行う事ができるため、記述量が低減され、開発期間の短縮のみならず品質向上にも寄与する。

また、一般のクロック境界を指定しないプログラム・レベルでの記述では表現できない、バス・インターフェース回路や調停回路の記述が可能となる。特に、レジスタ代入が記述可能である為、ステートメントレベルでの並列性を考慮した記述を行う事が可能であり、ストール動作を伴うパイプライン動作のような複雑な回路動作をＣ記述よりも少ないコード量で容易に記述可能である。

また、一般のＣコンパイラでコンパイル可能なＣ記述へ変換する為、高速なシミュレーションが可能となり、機能検証工数の大幅な低減が可能となる。従って、機能設計における論理設計、論理検証の双方の大幅な工数削減が可能となる。

高位合成ツールが不得意とする、サイクル精度を要求される例えば、キャッシュ・コントローラやＤＭＡコントローラの開発に適用可能であり、設計期間の短縮に大きく寄与する。

以上本発明者によってなされた発明を実施形態に基づいて具体的に説明したが、本発明はそれに限定されるものではなく、その要旨を逸脱しない範囲において種々変更可能であることは言うまでもない。

例えば、以上説明したプログラム記述及び回路記述は一例であり種々

の論理設計に適用することができる。ＨＤＬは必ずしもＲＴＬに限定されない。プログラム記述言語はＣ言語に限定されず、その他の高級言語であってもよい。更にＪａｖａ（登録商標）等の仮想マシン言語などを用いることも可能である。

5

産業上の利用可能性

本発明は、ＣＰＵなどの論理回路の設計に広く適用することができる。

請 求 の 範 囲

1．所定のプログラム言語を流用して記述された第1プログラム記述を回路記述に変換可能なコンパイラであって、

前記第1プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含み、

前記回路記述は、前記第1プログラム記述が特定する回路動作を実現するハードウェアを所定のハードウェア記述言語で特定することを特徴とするコンパイラ。

2．所定のプログラム言語を流用して記述された第1プログラム記述を所定のプログラム言語を用いた第2プログラム記述に変換可能なコンパイラであって、

前記第1プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含み、

前記第2プログラム記述は、前のサイクルの状態を参照可能にする為に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文とを含むことを特徴とするコンパイラ。

3．所定のプログラム言語を流用して記述された第1プログラム記述を、所定のプログラム言語を用いた第2プログラム記述と回路記述に変換可能なコンパイラであって、

前記第1プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含み、

前記第2プログラム記述は、前のサイクルの状態を参照可能にする為に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述

に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文とを含み、

前記回路記述は、前記第２プログラム記述で定義されるハードウェアを所定のハードウェア記述言語で特定することを特徴とするコンパイラ。

４．前記所定のプログラム言語はＣ言語であることを特徴とする請求項１乃至３の何れか１項記載のコンパイラ。

５．前記ハードウェア記述言語はＲＴＬレベルの記述言語であることを特徴とする請求項１又は３記載のコンパイラ。

６．タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含む第１プログラム記述を入力する第１処理と、

前記第１プログラム記述に基づいて前記タイミング仕様を満足する回路情報を生成する第２処理と、を含むことを特徴とする論理回路の設計方法。

７．前記第２処理は、前記第１プログラム記述を変換して、レジスタ代入文が入力変数と出力変数を用いて変形されると共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含む第２プログラム記述を前記回路情報として生成する処理を含むことを特徴とする請求項６記載の論理回路の設計方法。

８．前記第２処理は、前記第２プログラム記述を変換して、前記タイミング仕様を満足するハードウェアを所定のハードウェア記述言語で特定するための回路記述を更に別の前記回路情報として生成する処理を含むことを特徴とする請求項７記載の論理回路の設計方法。

９．前記プログラム言語はＣ言語であることを特徴とする請求項８記載

の論理回路の設計方法。

１０．前記第２プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含むことを特徴とする請求項９記載の論理回路の設計方法。

１１．前記第２処理は、前記第１プログラム記述を変換して、前記レジスタ代入文が入力変数と出力変数を用いて変形された記述を含む第２プログラム記述を前記回路情報として生成する処理を含むことを特徴とする請求項６記載の論理回路の設計方法。

１２．前記第２処理は、前記第２プログラム記述を変換して、前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含み、所定のプログラム言語で記述されてコンピュータで実行可能なむ第３プログラム記述を、前記回路情報として生成する処理を含むことを特徴とする請求項１１記載の論理回路の設計方法。

１３．前記第３プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含むことを特徴とする請求項１２記載の論理回路の設計方法。

１４．タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含む第１プログラム記述を入力する入力処理と、

前記レジスタ代入文が入力変数と出力変数を用いて変形されると共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含み、前記所定のプログラム言語で記述された第２プログラム記述を生成する変換処理と、を含むことを特徴とする論理回路の設計方法。

１５．前記変換処理は、第１プログラム記述に基づいてＣＦＧを生成す

る過程で、前記ＣＦＧに前記クロック境界記述に対応してクロック境界ノードを設定し、前記クロック境界ノードの後に、前記レジスタ代入記述を挿入することを特徴とする請求項１４記載の論理回路の設計方法。

１６.第２プログラム記述に対してそのＣＦＧを利用しながらステート遷移毎の変数表を作成しながらコード最適化を行う最適化処理を更に含むことを特徴とする請求項１５記載の論理回路の設計方法。

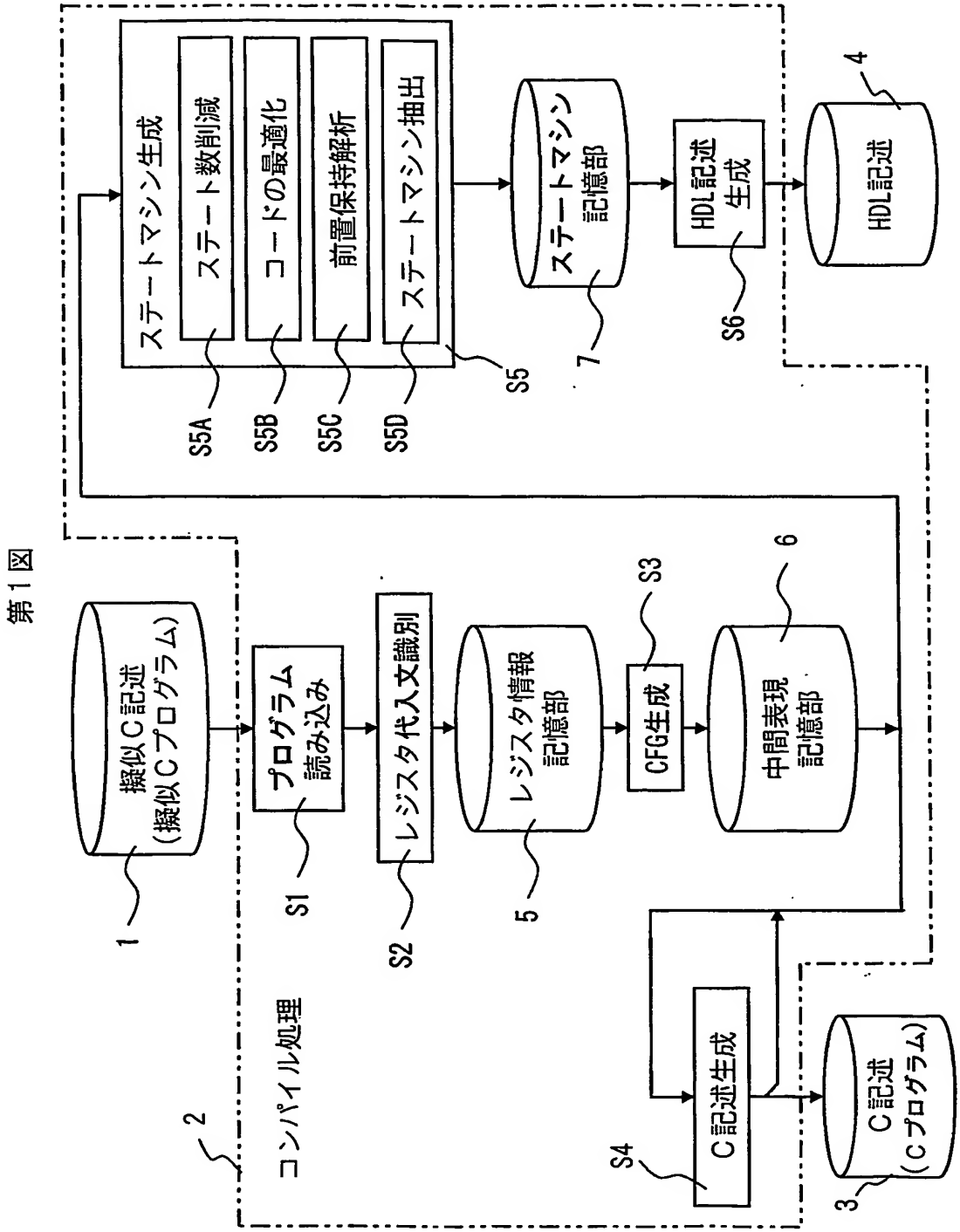
１７.前記変数表においてステート間で変数に変化のない部分を前置保持を要する部分として抽出し、抽出された部分に、出力変数に入力変数を代入する記述を追加する前置保持処理を更に含むことを特徴とする請求項１６記載の論理回路の設計方法。

１８.前記前置保持処理を経た変数表の各ステート遷移毎の変数と引数に基づいてステートマシンを構成するコードの抽出を行う抽出処理を更に含むことを特徴とする請求項１７記載の論理回路の設計方法。

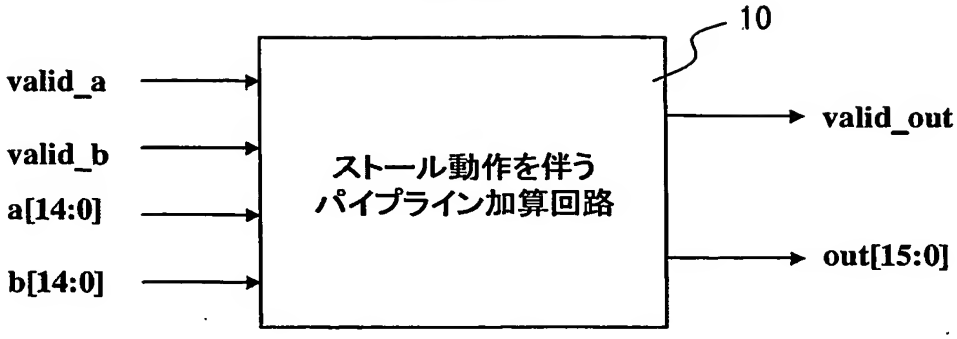
１９.前記抽出処理で抽出されたステートマシン構成コードと第２プログラム記述を参照しながら、前記回路仕様を満足する回路のハードウェアを所定のハードウェア記述言語で記述する処理を更に含むことを特徴とする請求項１８記載の論理回路の設計方法。

２０.前記第１プログラム記述に対して０サイクルで実行されるループが存在するか否かが判定され、存在しないと判別されたときに前記回路仕様を満足する回路のハードウェアを所定のハードウェア記述言語で記述する処理を行うことを特徴とする請求項１４記載の論理回路の設計方法。

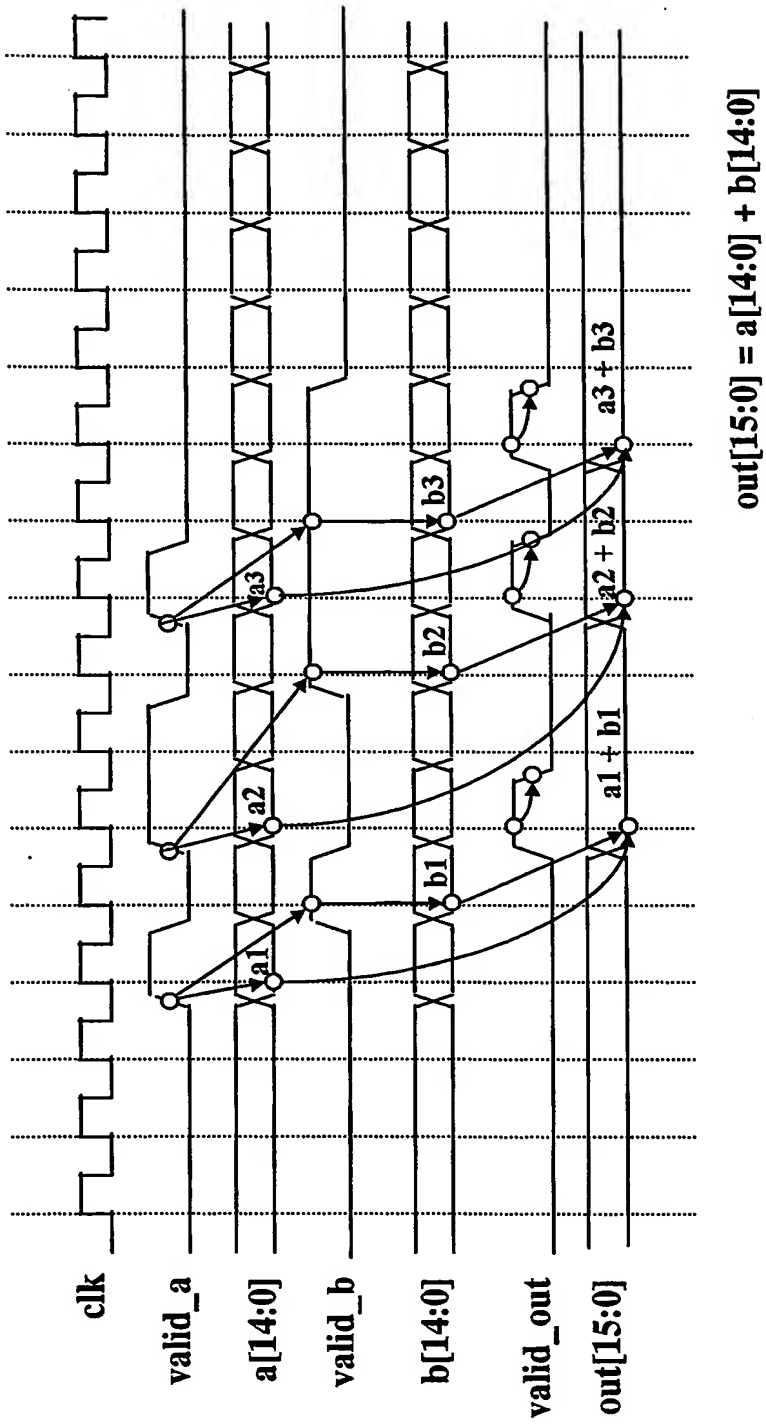
第1図


擬似C記述
(擬似Cプログラム)
1
プログラム
読み込み
S1
レジスタ代入文識別
S2
レジスタ情報
記憶部
5
CFG生成
S3
中間表現
記憶部
6
ステートマシン生成
ステート数削減
コードの最適化
前置保持解析
ステートマシン抽出
S5A
S5B
S5C
S5D
S5
ステートマシン
記憶部
7
HDL記述
生成
S6
HDL記述
4
C記述生成
S4
C記述
(Cプログラム)
3
コンパイル処理
2

第２図


10
valid_a
valid_b
a[14:0]
b[14:0]
ストール動作を伴う
パイプライン加算回路
valid_out
out[15:0]

第３図


clk
valid_a
a[14:0]
valid_b
b[14:0]
valid_out
out[15:0]
a1
a2
a3
b1
b2
b3
a1 + b1
a2 + b2
a3 + b3
out[15:0] = a[14:0] + b[14:0]

第4図

```
1  #include <stdio.h>
2  void pipeline(unsigned short valid_a,
3                unsigned short valid_b,
4                unsigned short a,
5                unsigned short b,
6                unsigned short *out,
7                unsigned short *valid_out);
8  main() {
9    unsigned short valid_a, valid_b,
10                  a, b, *out, *valid_out;
11   *out       = 0x0000;
12   *valid_out = 0x0000;
13   pipeline(valid_a, valid_b, a, b, out, valid_out);
14 }
```

```
15 void pipeline(unsigned short valid_a, unsigned short valid_b,
16               unsigned short a, unsigned short b,
17               unsigned short *out, unsigned short *valid_out) {
18   unsigned short valid_a_tmp = 0x0000;
19   unsigned short a_tmp = 0x0000;
20   unsigned short b_tmp = 0x0000;
21   while (1) {
22     valid_a_tmp = $ 0x0001&valid_a;
23     if ((0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)) {
24       a_tmp = 0x7FFF&a;
25       $
26     L:
27       if (0x0001&valid_b == 0x0001) b_tmp = 0x7FFF&b;
28       else                          $ goto L;
29       *out = $ (a_tmp + b_tmp);
30       *valid_out = $ 0x0001;
31     } else {
32       $
33       *valid_out = $ 0x0000;
34     }
35   }
36 }
```

11(回路動作記述部)

第５図

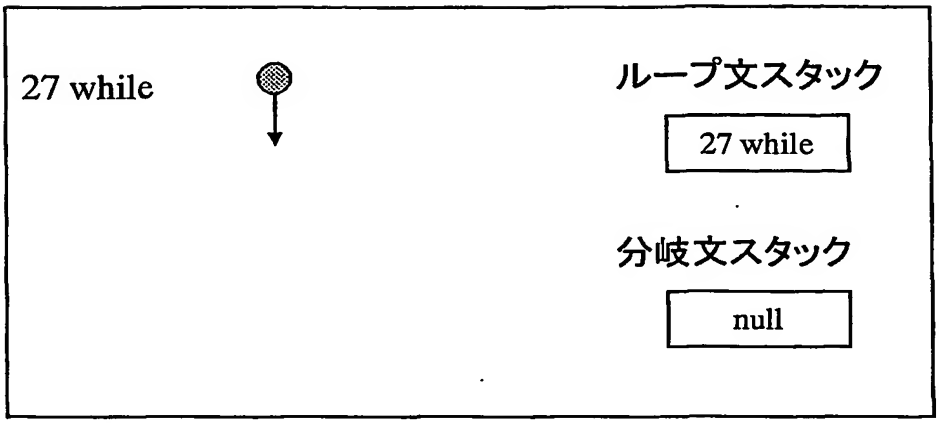
```
1  #include <stdio.h>
2  void pipeline(unsigned short valid_a,
3                unsigned short valid_b,
4                unsigned short a,
5                unsigned short b,
6                unsigned short *out,
7                unsigned short *valid_out);
8  main() {
9    unsigned short valid_a, valid_b,
10                  a, b, *out, *valid_out;
11   *out       = 0x0000;
12   *valid_out = 0x0000;
13   pipeline(valid_a, valid_b, a, b, out, valid_out);
14 }
```

追加変数宣言

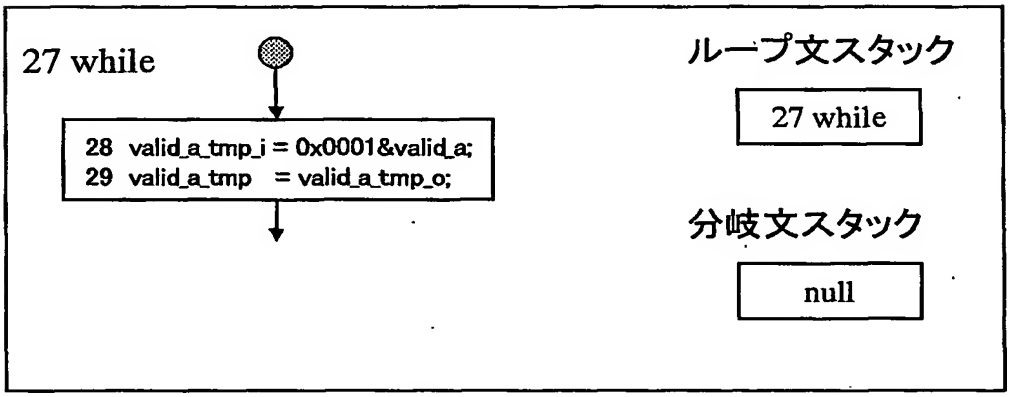
13(レジスタ代入文書き換え文)

```
15 void pipeline(unsigned short valid_a, unsigned short valid_b,
16               unsigned short a, unsigned short b,
17               unsigned short *out, unsigned short *valid_out) {
18   unsigned short valid_a_tmp = 0x0000;
19   unsigned short a_tmp = 0x0000;
20   unsigned short b_tmp = 0x0000;
21   unsigned short valid_a_tmp_i;
22   unsigned short valid_a_tmp_o = 0x0000;
23   unsigned short out_i;
24   unsigned short out_o = 0x0000;
25   unsigned short valid_out_i;
26   unsigned short valid_out_o = 0x0000;
27   while (1) {
28     valid_a_tmp_i = 0x0001&valid_a;
29     valid_a_tmp   = valid_a_tmp_o;
30     if ((0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)) {
31       a_tmp = 0x7FFF&a;
32       $
33     L:
34       if (0x0001&valid_b == 0x0001) b_tmp = 0x7FFF&b;
35       else                          $ goto L;
36       out_i = (a_tmp + b_tmp);
37       *out = out_o;
38       valid_out_i = 0x0001;
39       *valid_out = valid_out_o;
40     } else {
41       $
42       valid_out_i = 0x0001;
43       *valid_out = valid_out_o;
45     }
46   }
47 }
```

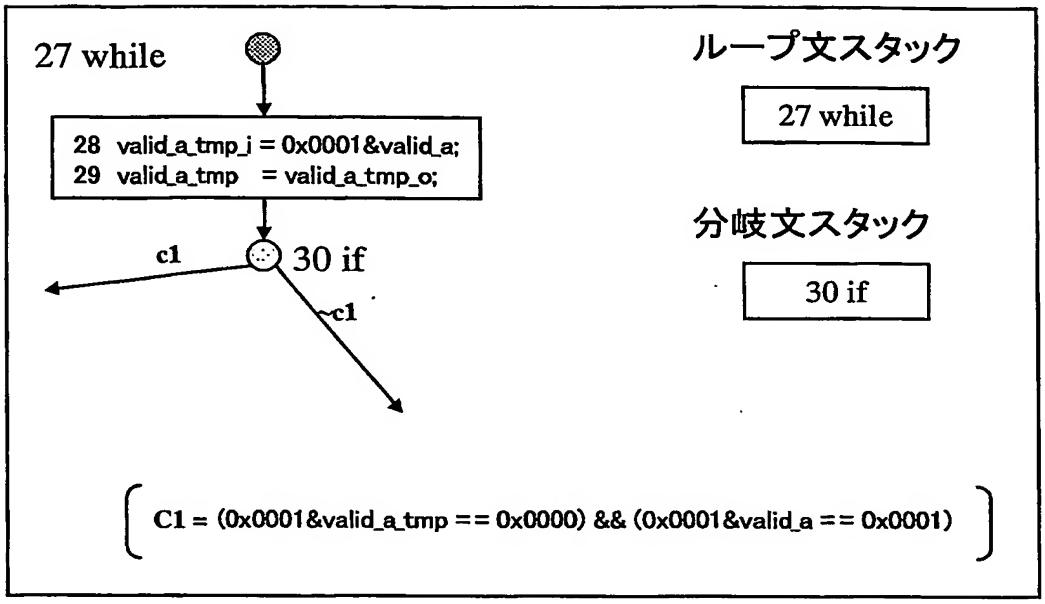
第6図


27 while
ループ文スタック
27 while
分岐文スタック
null

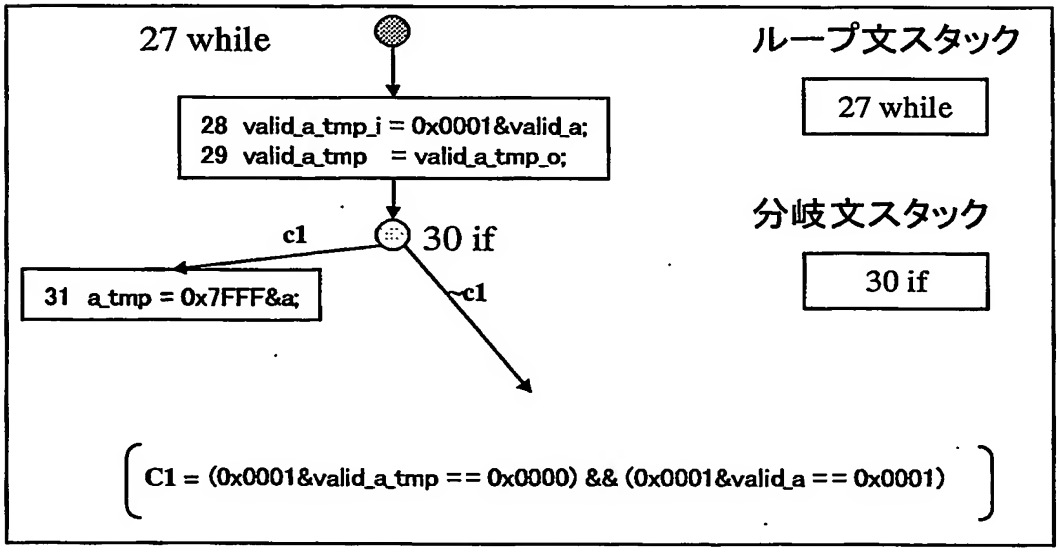
第7図


27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
ループ文スタック
27 while
分岐文スタック
null

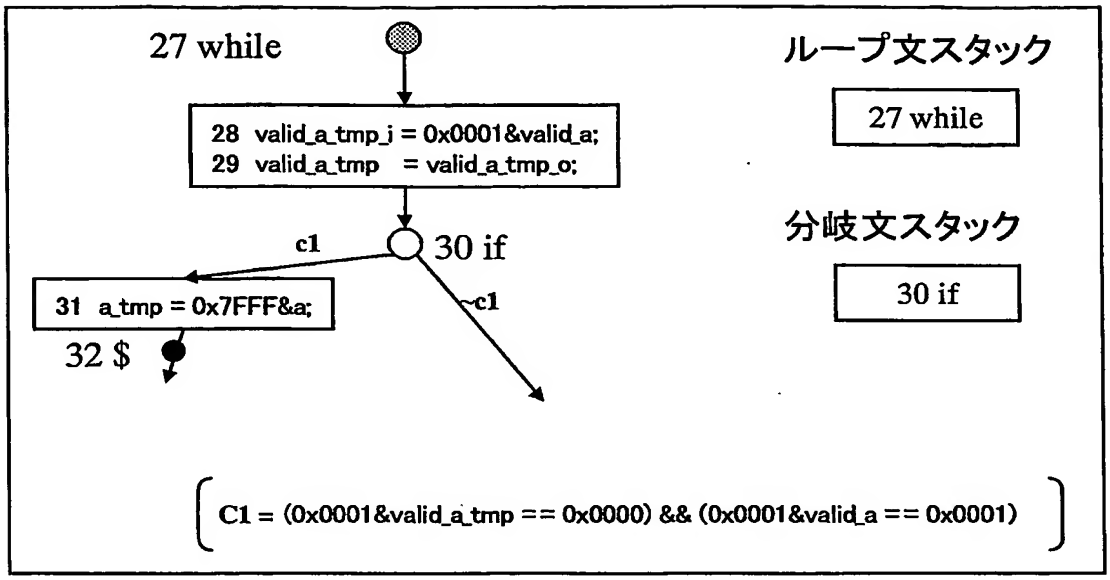
第８図


27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
30 if
~c1
ループ文スタック
27 while
分岐文スタック
30 if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

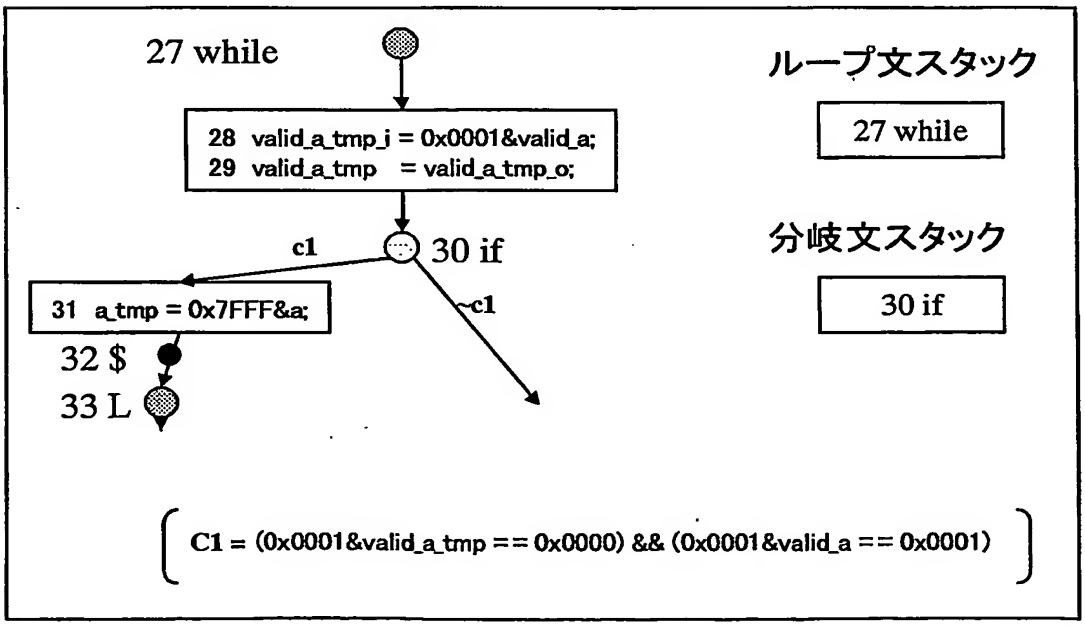
第９図


27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
30 if
31 a_tmp = 0x7FFF&a;
~c1
ループ文スタック
27 while
分岐文スタック
30 if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

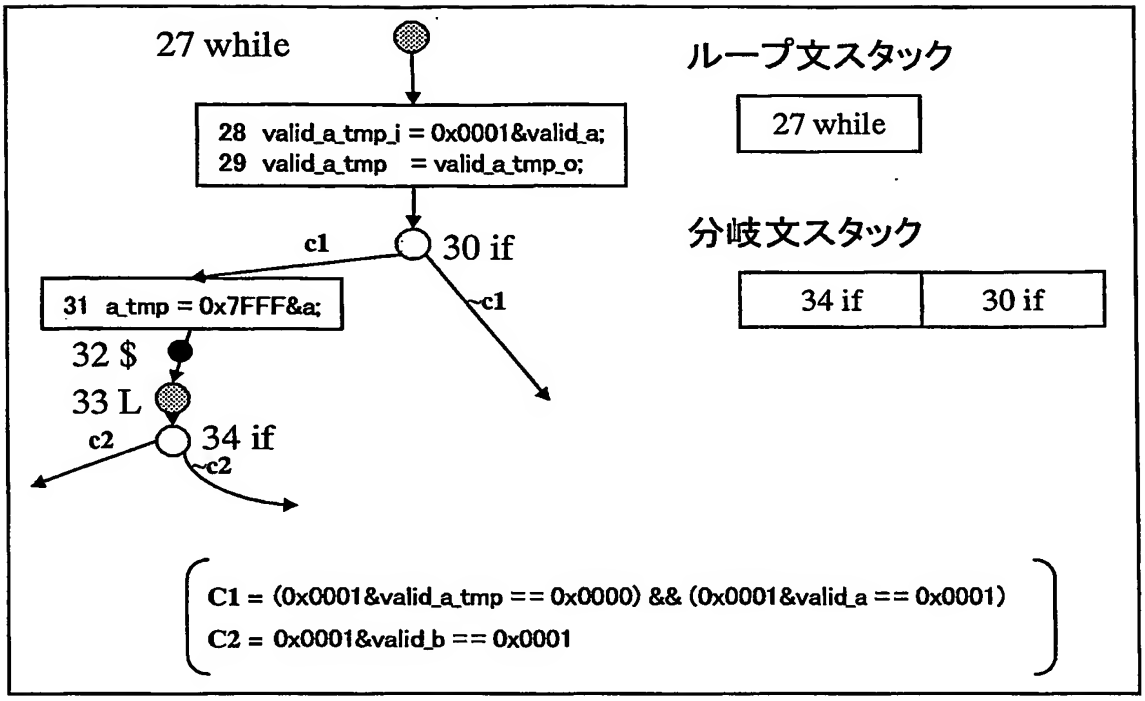
第１０図


27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
30 if
~c1
31 a_tmp = 0x7FFF&a;
32 $
ループ文スタック
27 while
分岐文スタック
30 if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

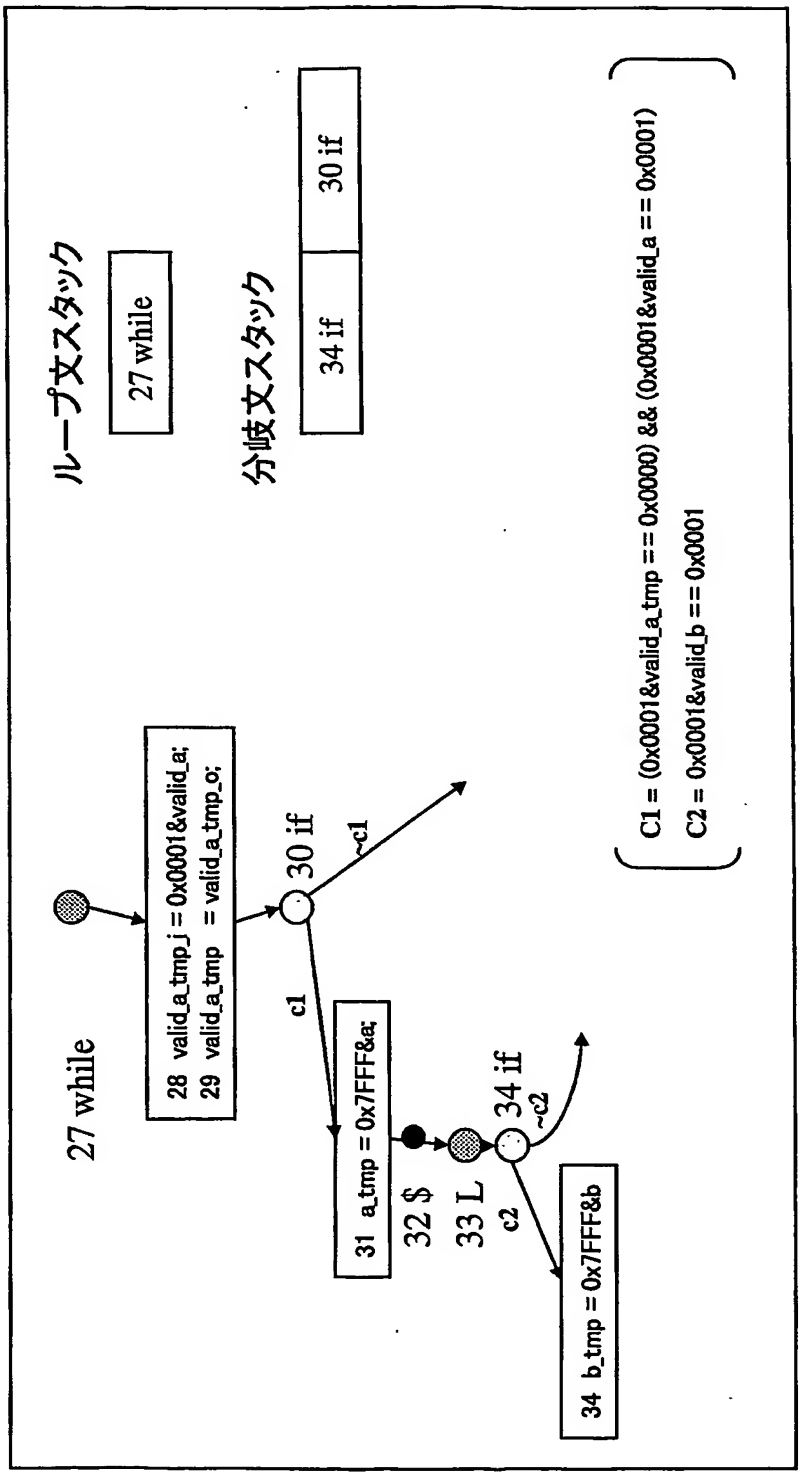
第１１図


27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
30 if
~c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
ループ文スタック
27 while
分岐文スタック
30 if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

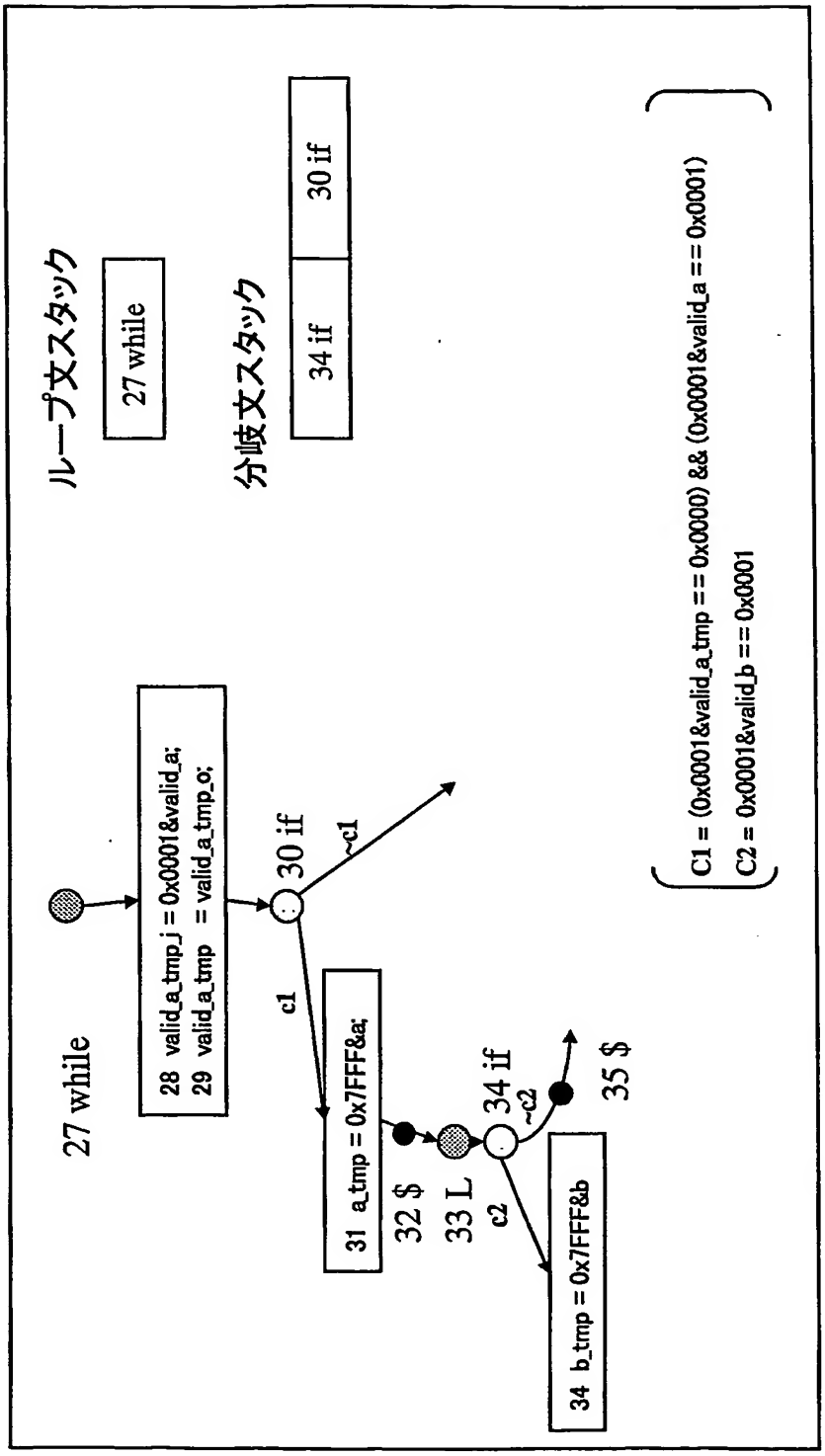
第１２図


27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
30 if
31 a_tmp = 0x7FFF&a;
~c1
32 $
33 L
c2
34 if
~c2
ループ文スタック
27 while
分岐文スタック
34 if
30 if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

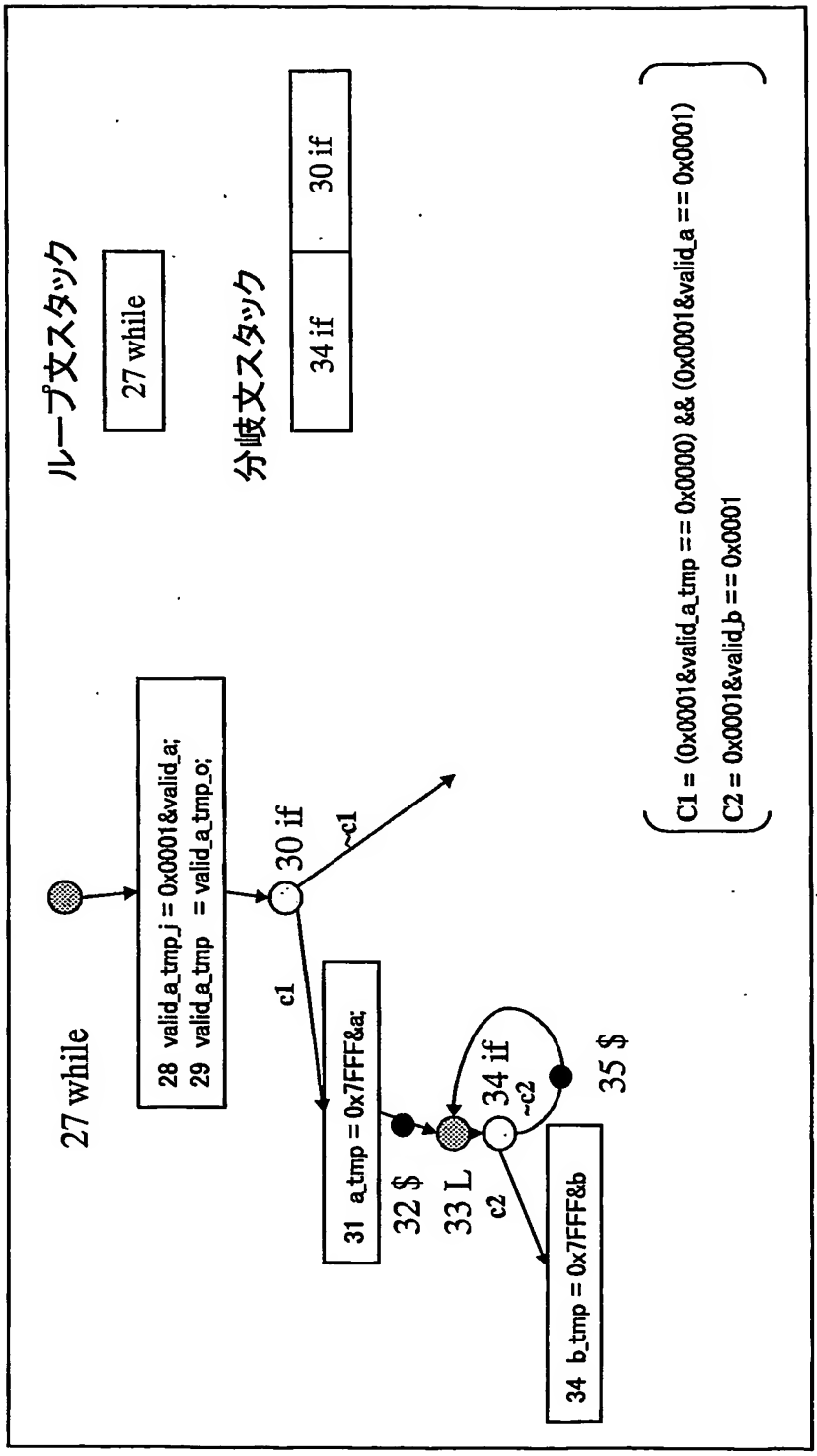
第１３図


ループ文スタック
27 while
分岐文スタック
34 if
30 if
27 while
28 valid_a_tmp:j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
~c1
c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
34 b_tmp = 0x7FFF&b
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

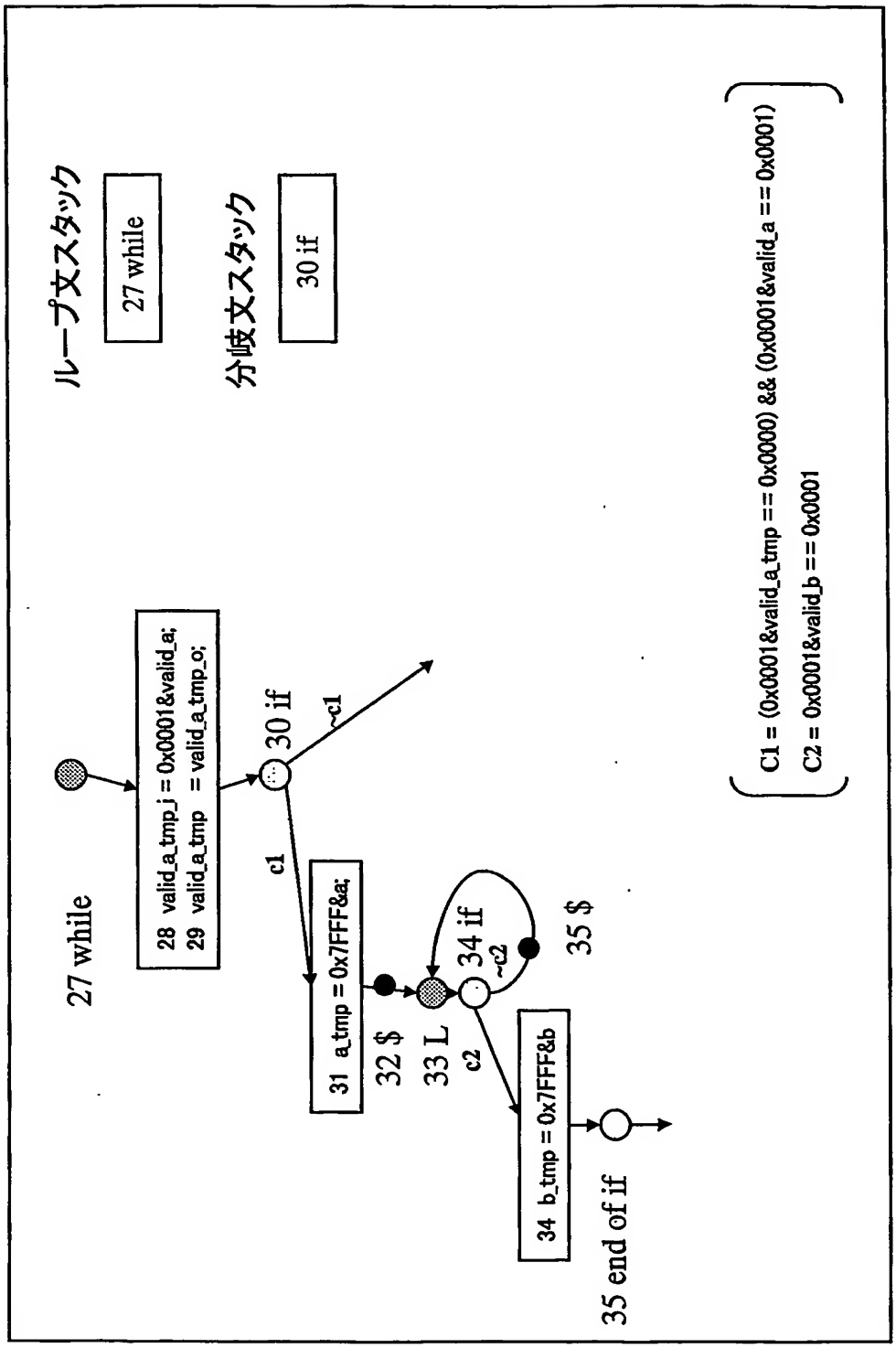
第14図


ループ文スタック
27 while
分岐文スタック
34 if
30 if
27 while
28 valid_a_tmp.j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
c2
34 if
~c2
35 $
34 b_tmp = 0x7FFF&b
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

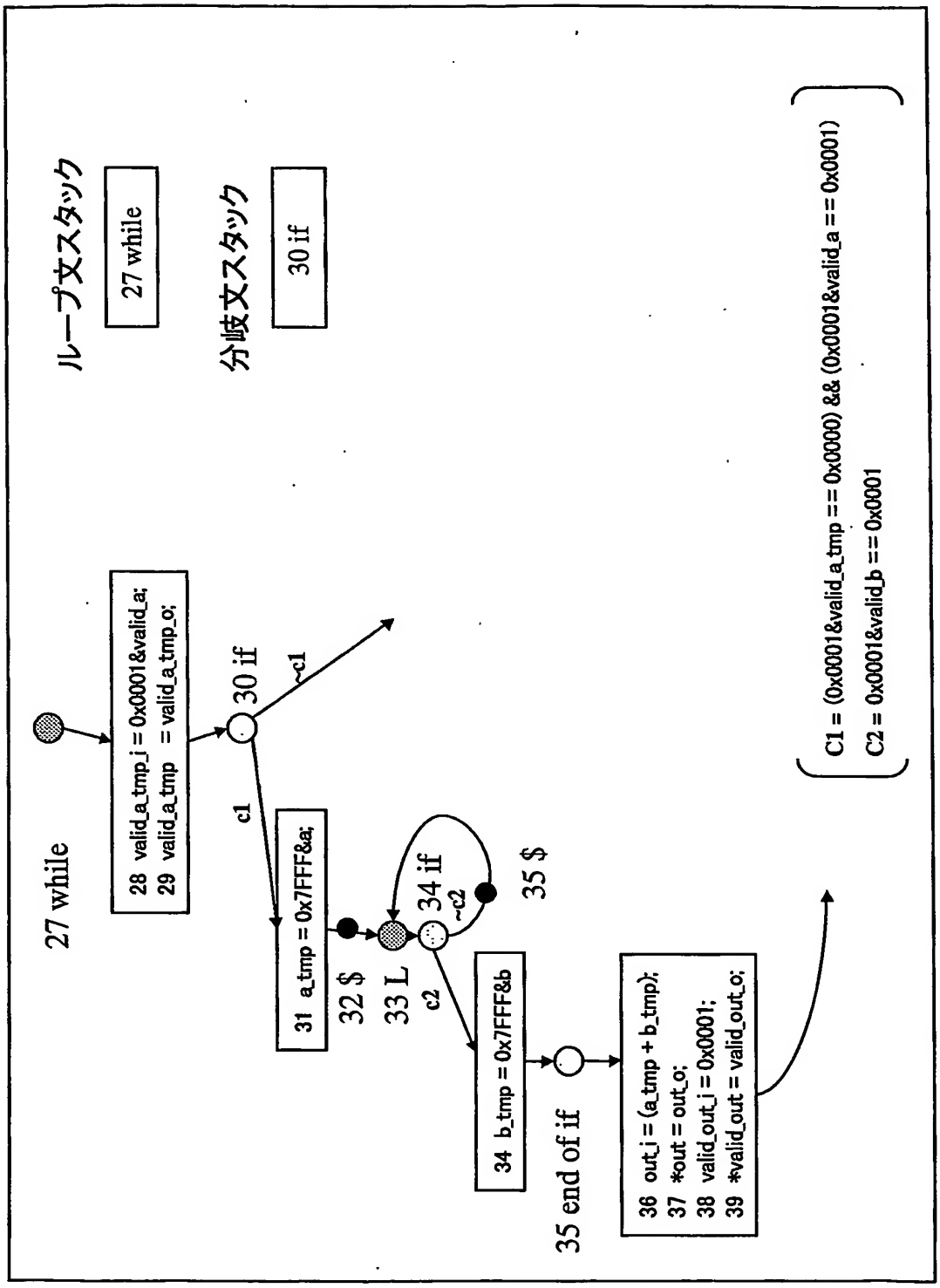
第15図


ループ文スタック
27 while
分岐文スタック
34 if
30 if
27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
~c1
c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
34 b_tmp = 0x7FFF&b
35 $
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

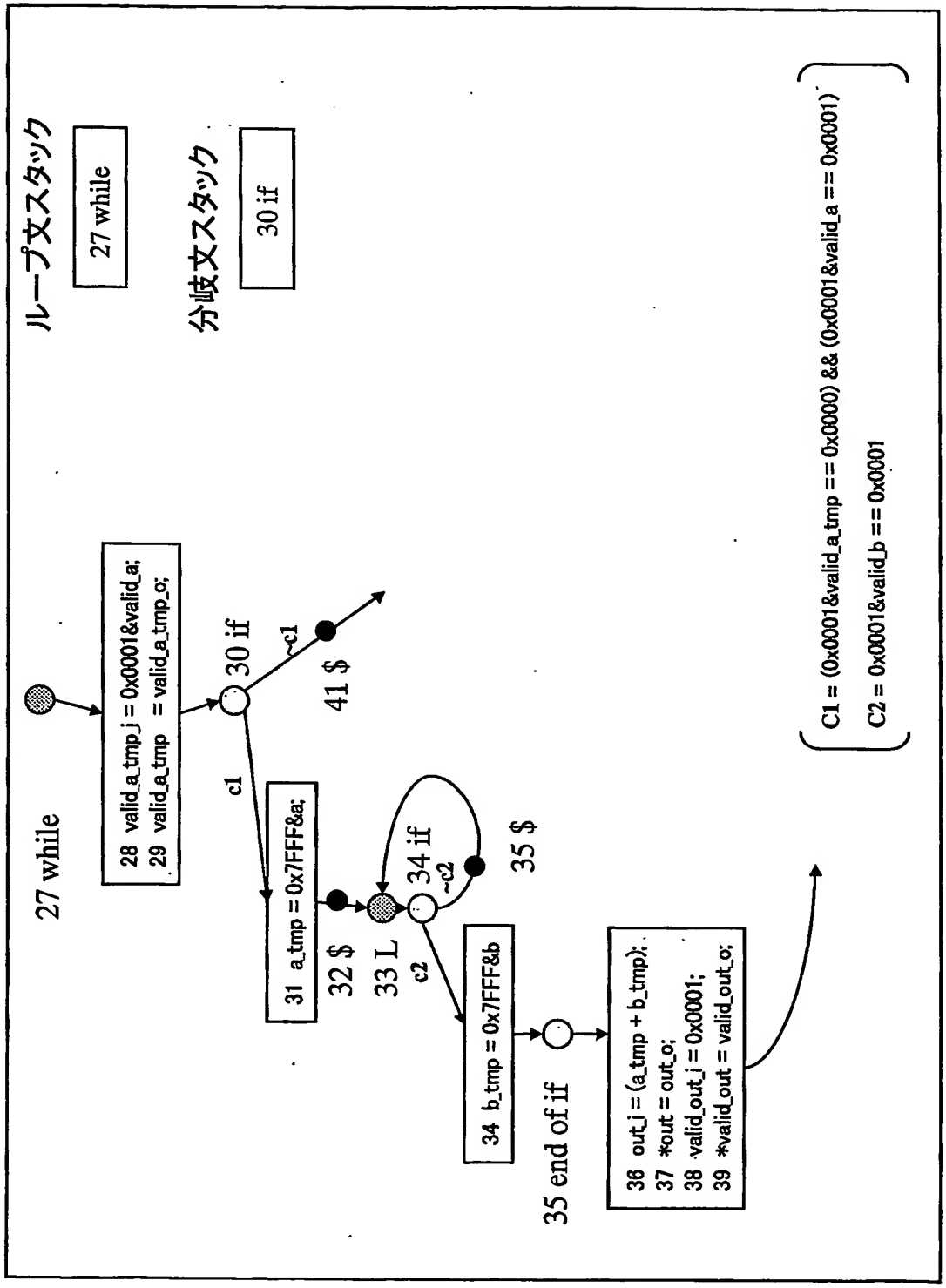
第16図


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
~c1
c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

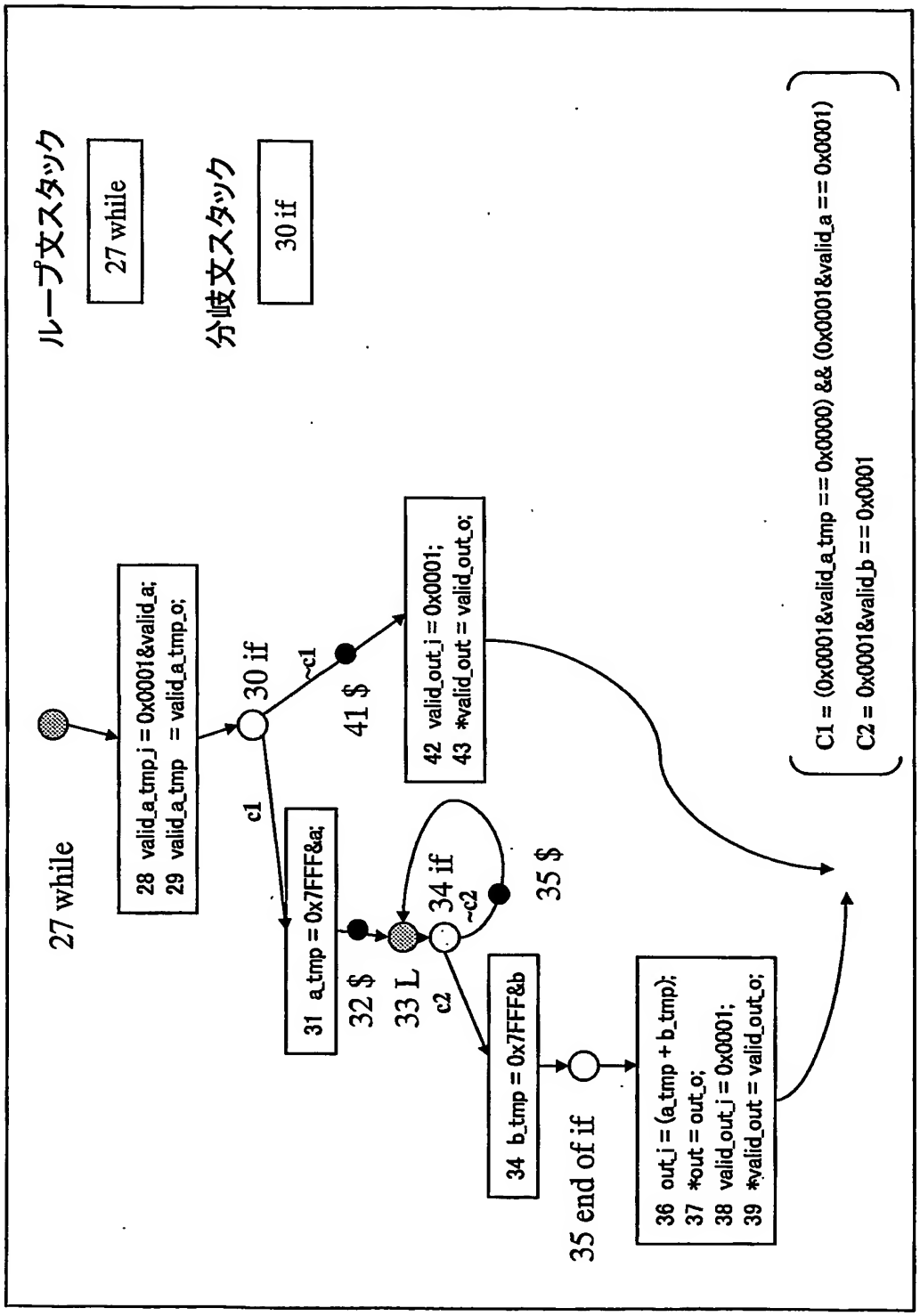
第17図


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
c2
~c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

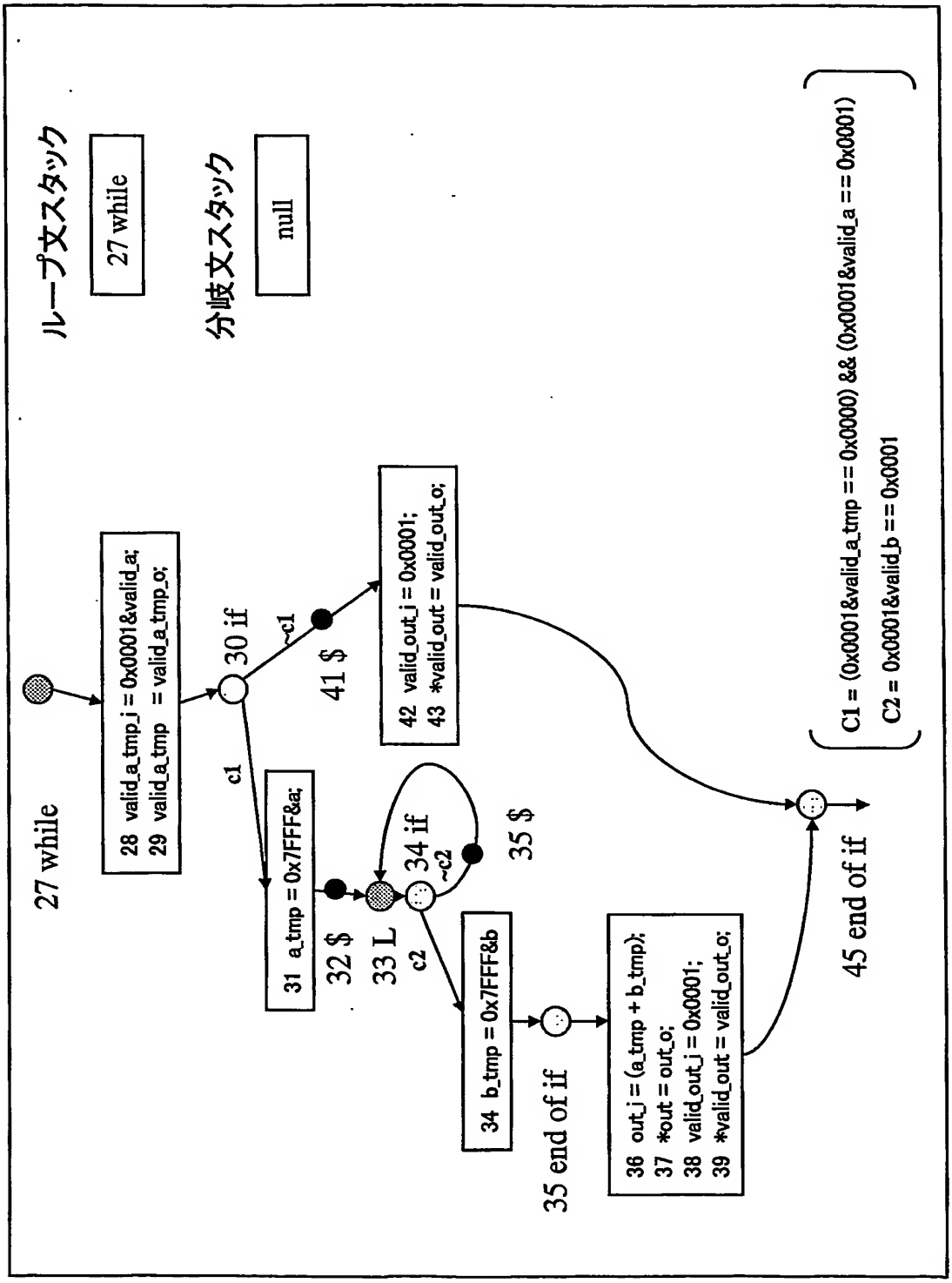
第１８図


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
41 $
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

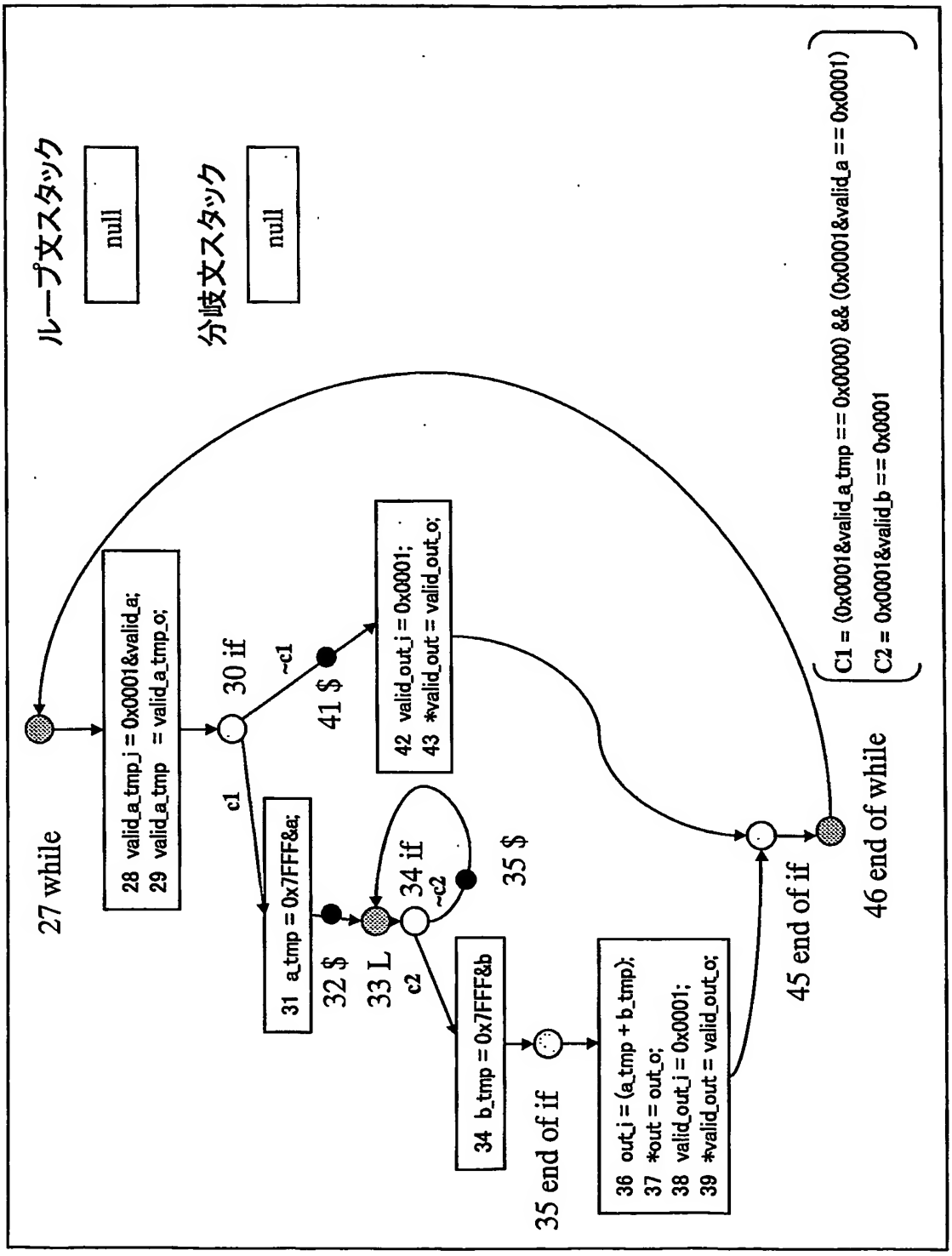
第19図


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
41 $
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

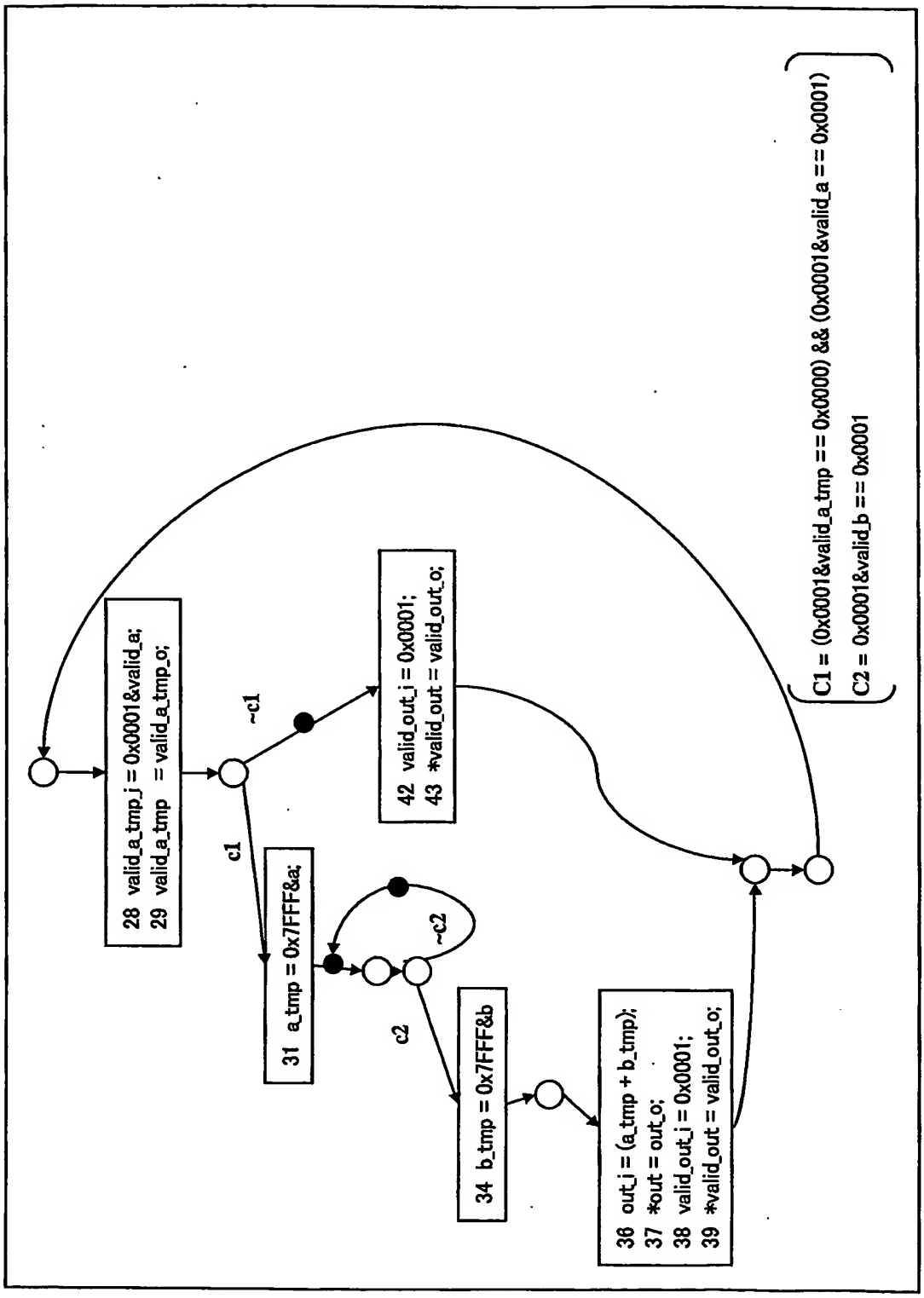
第２０図


ループ文スタック
27 while
分岐文スタック
null
27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
41 $
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
45 end of if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

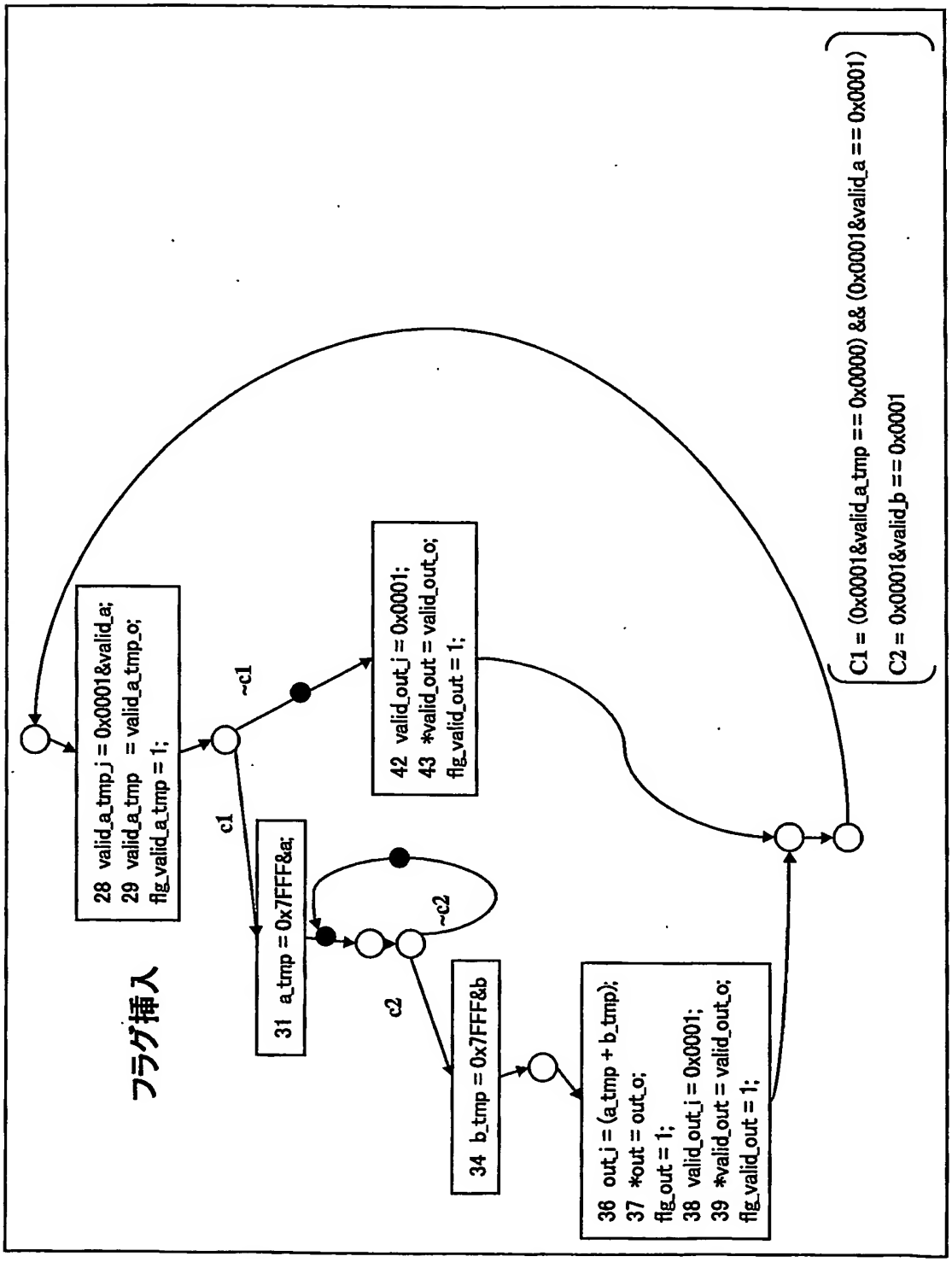
第２１図


ループ文スタック
null
分岐文スタック
null
27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
c2
~c2
34 b_tmp = 0x7FFF&b
35 $
35 end of if
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
41 $
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
45 end of if
46 end of while
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

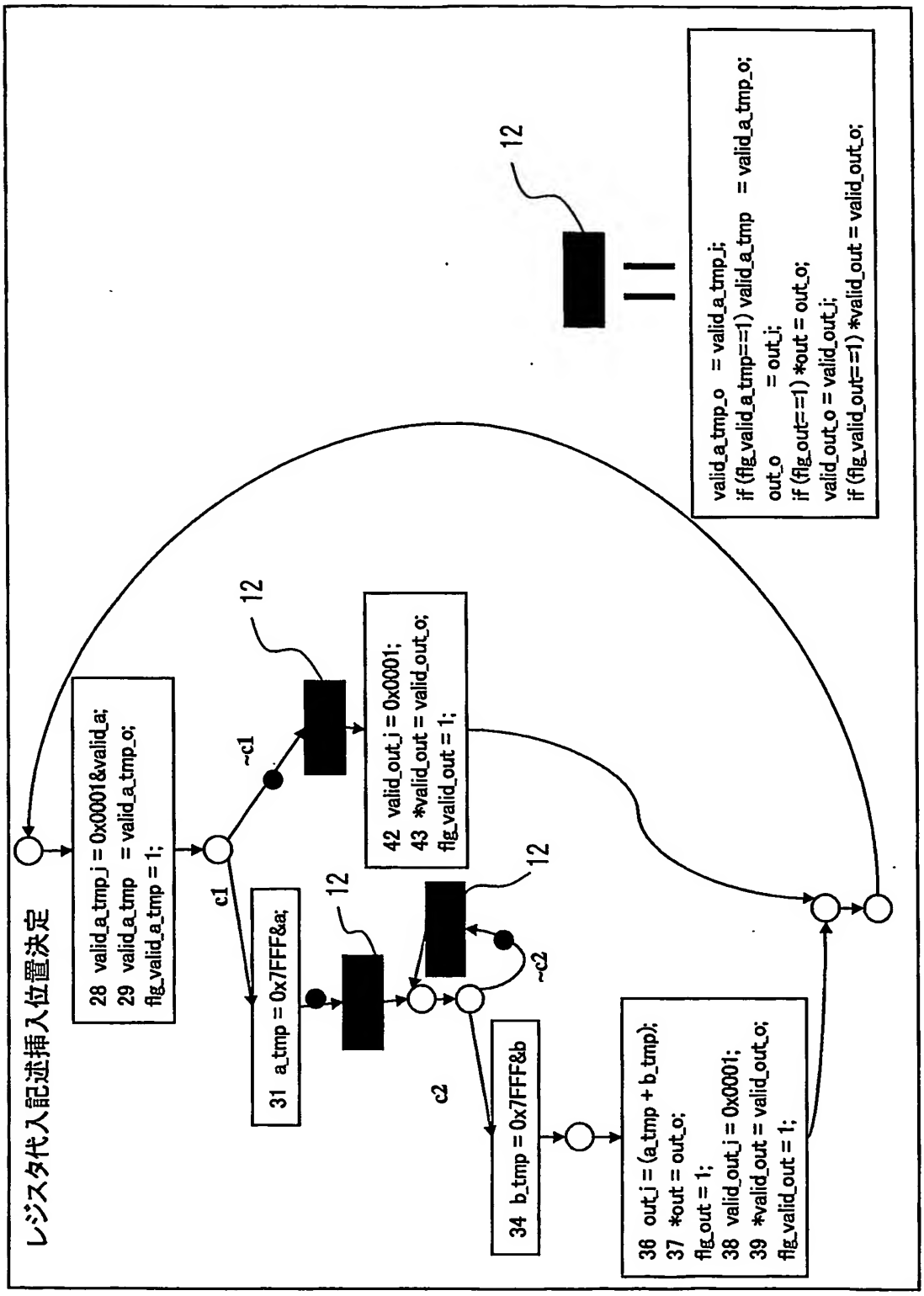
第22図


28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
~c1
c1
31 a_tmp = 0x7FFF&a;
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
~c2
c2
34 b_tmp = 0x7FFF&b
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

第２３図


フラグ挿入
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
flg_valid_a_tmp = 1;
~c1
c1
31 a_tmp = 0x7FFF&a;
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
flg_valid_out = 1;
~c2
c2
34 b_tmp = 0x7FFF&b
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
flg_out = 1;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
flg_valid_out = 1;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

第２４図


レジスタ代入記述挿入位置決定
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
flg_valid_a_tmp = 1;
c1
~c1
12
31 a_tmp = 0x7FFF&a;
12
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
flg_valid_out = 1;
12
c2
~c2
34 b_tmp = 0x7FFF&b
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
flg_out = 1;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
flg_valid_out = 1;
12
valid_a_tmp_o = valid_a_tmp_i;
if (flg_valid_a_tmp==1) valid_a_tmp = valid_a_tmp_o;
out_o = out_i;
if (flg_out==1) *out = out_o;
valid_out_o = valid_out_i;
if (flg_valid_out==1) *valid_out = valid_out_o;

第２５図

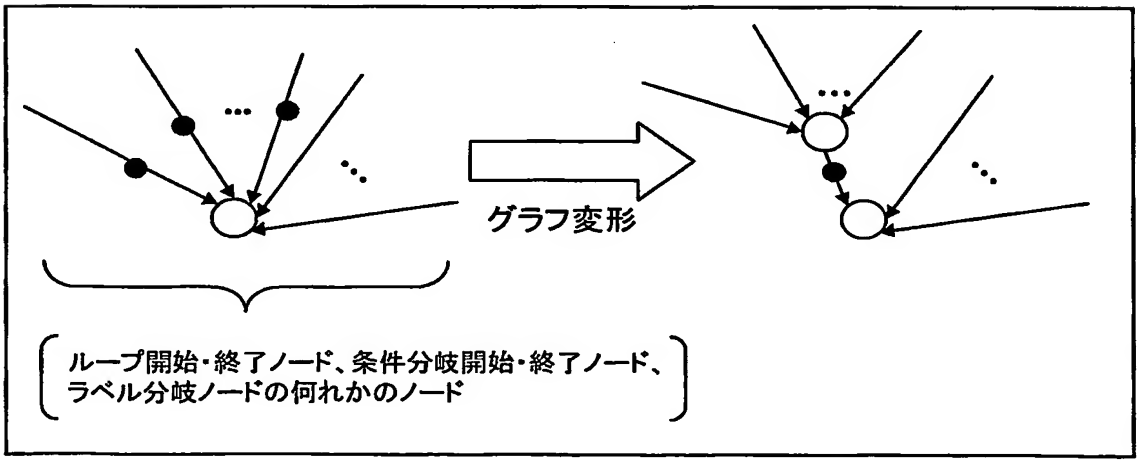
```
1  #include <stdio.h>
2  void pipeline(unsigned short valid_a,
3                unsigned short valid_b,
4                unsigned short a,
5                unsigned short b,
6                unsigned short *out,
7                unsigned short *valid_out);
8  main() {
9    unsigned short valid_a, valid_b,
10                  a, b, *out, *valid_out;
11   *out       = 0x0000;
12   *valid_out = 0x0000;
13   pipeline(valid_a, valid_b, a, b, out, valid_out);
14 }
```

```
15 void pipeline(unsigned short valid_a, unsigned short valid_b,
16               unsigned short a, unsigned short b,
17               unsigned short *out, unsigned short *valid_out) {
18   unsigned short valid_a_tmp      = 0x0000;
19   unsigned short a_tmp            = 0x0000;
20   unsigned short b_tmp            = 0x0000;
     /* Added variables */
21   unsigned short valid_a_tmp_i;
22   unsigned short valid_a_tmp_o    = 0x0000;
23   unsigned short valid_out_i;
24   unsigned short valid_out_o   = 0x0000;
25   unsigned short out_i;
26   unsigned short out_o            = 0x0000;
27   unsigned short flg_valid_a_tmp = 0x0000;
28   unsigned short flg_valid_out = 0x0000;
29   unsigned short flg_out          = 0x0000;
```

第２６図

```
30  while (1) {
     /* valid_a_tmp = $ valid_a; */
31   valid_a_tmp_i = 0x0001&valid_a;        /* Refined */
32   valid_a_tmp   = valid_a_tmp_o;         /* Refined */
33   flg_valid_a_tmp = 1;
34   if ((0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)) {
35     a_tmp = 0x7FFF&a;
      /* $ */
      /* BEGIN : Register Assignment */
36    valid_a_tmp_o   = valid_a_tmp_i;
37    if (flg_value_a_tmp == 1) valid_a_tmp     = valid_a_tmp_o;
38    out_o           = out_i;
39    if (flg_out==1) *out = out_o;
40    valid_out_o = valid_out_i;
41    if (flg_valid_out==1) *valid_out = valid_out_o;
      /* END   : Register Assignment */
42   L :
```

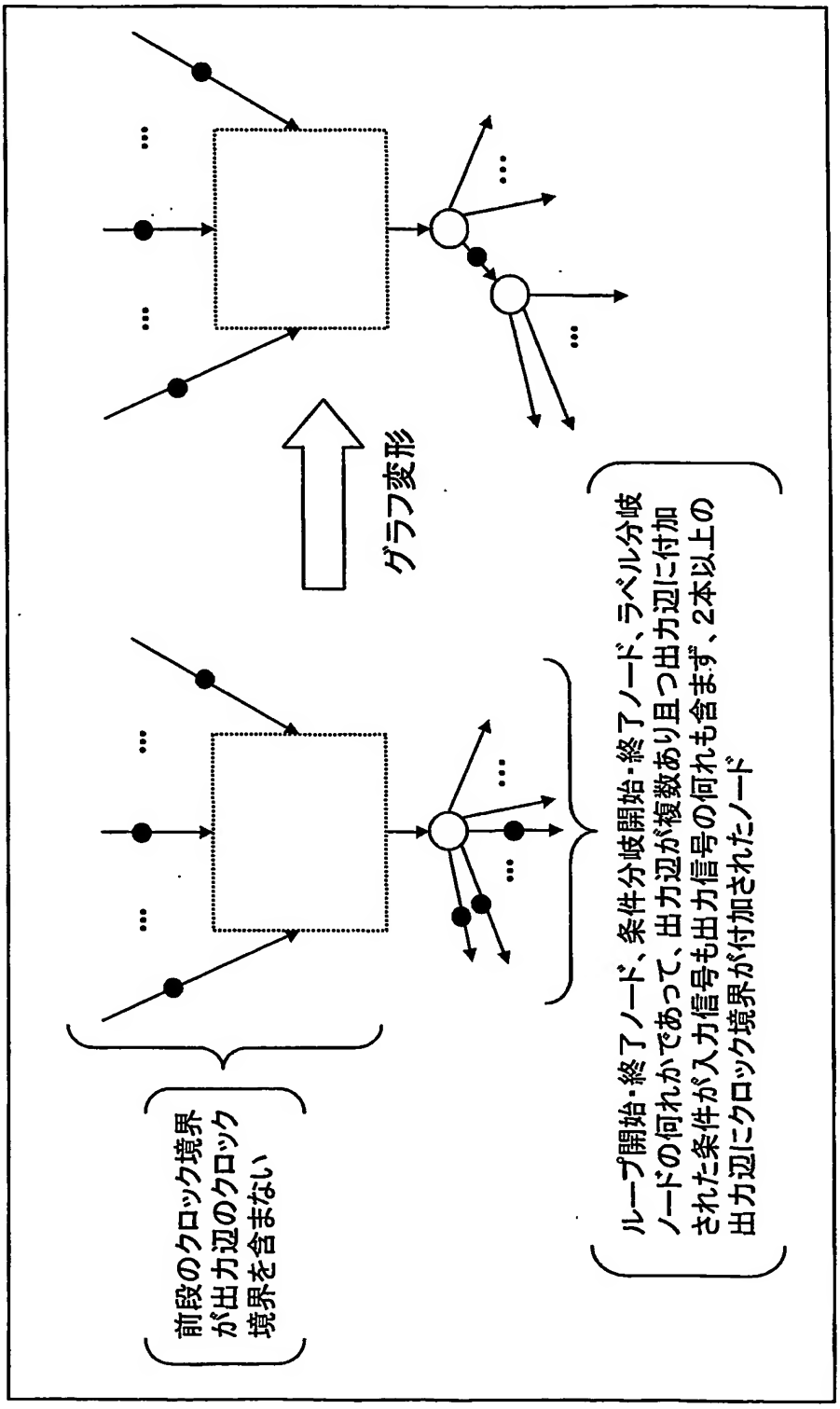
第２８図



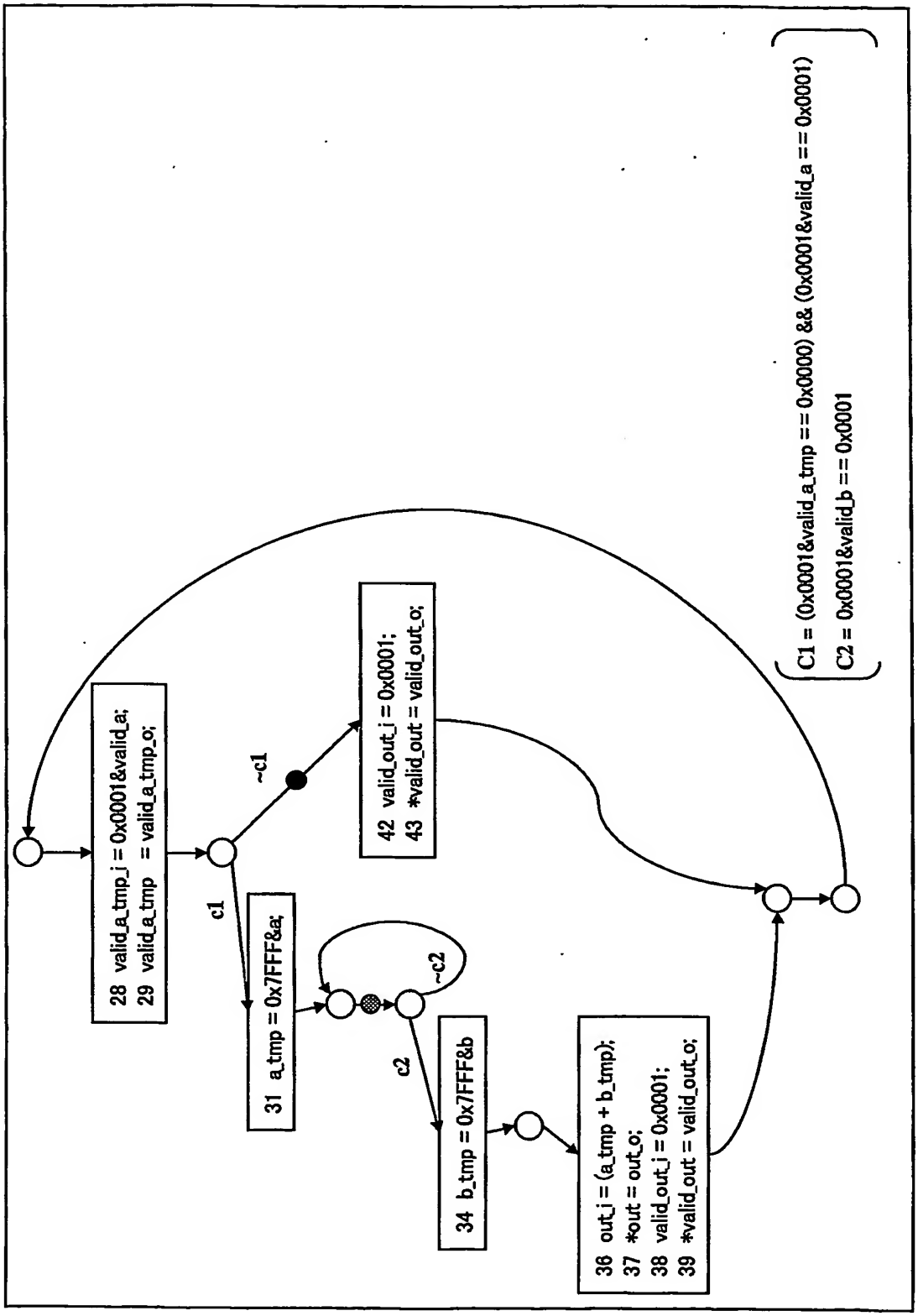
第２７図

```
43    if (0x0001&valid_b == 0x0001) b_tmp = 0x7FFF&b;
44    else {
       /* $ */
       /* BEGIN : Register Assignment */
45      valid_a_tmp_o   = valid_a_tmp_i;
46      valid_a_tmp     = valid_a_tmp_o;
47      out_o           = out_i;
48      if (flg_out==1) *out = out_o;
49      valid_out_o = valid_out_i;
50      if (flg_valid_out==1) *valid_out = valid_out_o;
       /* END   : Register Assignment */
51      goto L;
52    }
      /* *out = $ (a_tmp + b_tmp); */
53    out_i     = a_tmp + b_tmp;          /* Refined */
54    *out          = out_o;              /* Refined */
55    flg_out       = 1;                  /* Added */
      /* *valid_out = $ 0x0001; */
56    valid_out_i     = 0x0001;           /* Refined */
57    *valid_out          = valid_out_o; /* Refined */
58    flg_valid_out   = 1;              /* Added */
59  } else {
       /* $ */
       /* BEGIN : Register Assignment */
60      valid_a_tmp_o   = valid_a_tmp_i;
61      valid_a_tmp     = valid_a_tmp_o;
62      out_o       = out_i;
63      if (flg_out==1) *out = out_o;
64      valid_out_o = valid_out_i;
65      if (flg_valid_out==1) *valid_out = valid_out_o;
       /* END   : Register Assignment */
       /* *valid_out = $ 0x0000; */
66      valid_out_i = 0x0000;             /* Refined */
67      *valid_out      = valid_out_o;    /* Refined */
68      flg_valid_out   = 1;              /* Added */
69    }
70  }
71 }
```

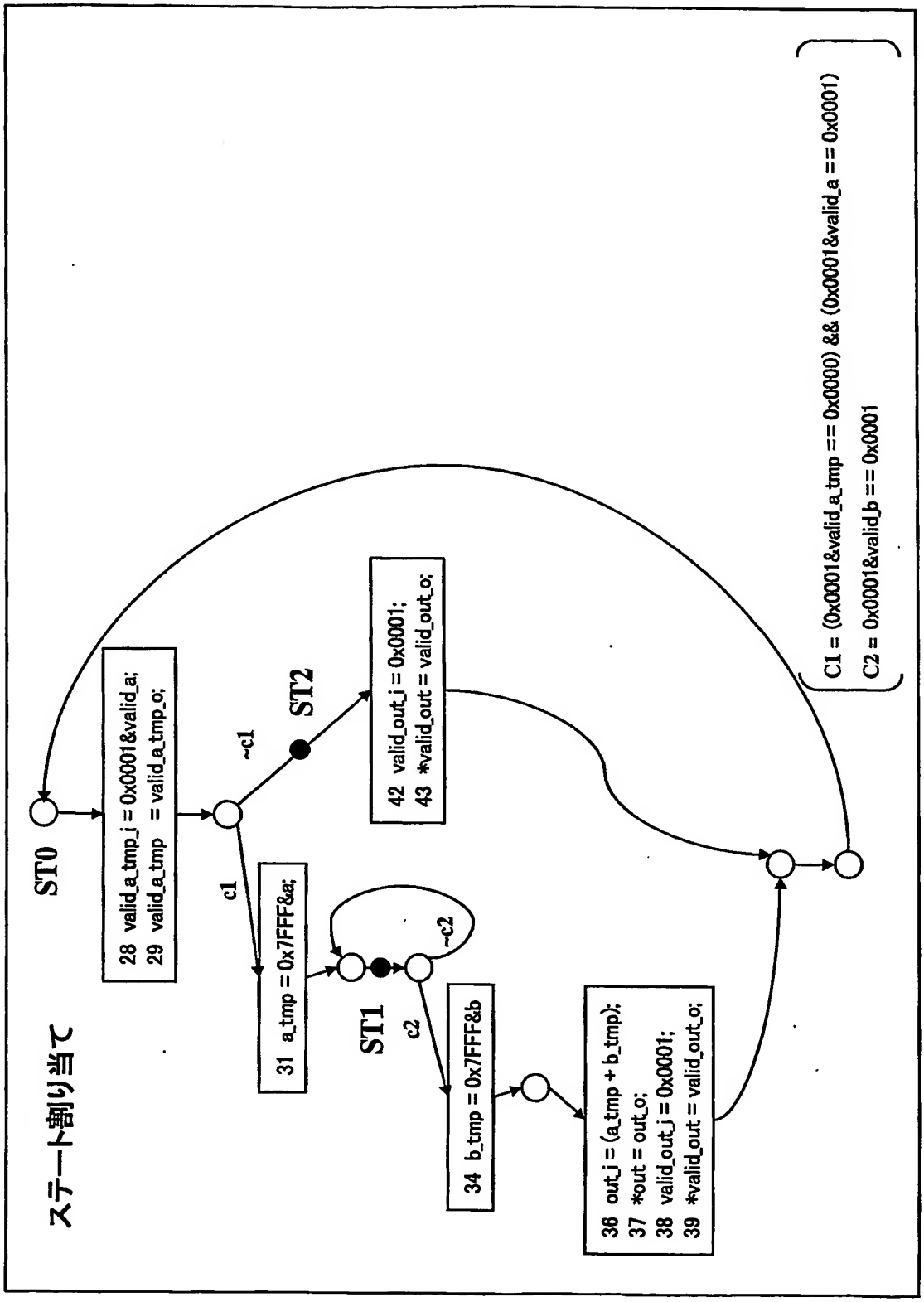
第２９図


前段のクロック境界が出力辺のクロック境界を含まない
グラフ変形
ループ開始・終了ノード、条件分岐開始・終了ノード、ラベル分岐ノードの何れかであって、出力辺が複数あり且つ出力辺に付加された条件が入力信号も出力信号の何れも含まず、2本以上の出力辺にクロック境界が付加されたノード

第３０図


28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
~c1
31 a_tmp = 0x7FFF&a;
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
~c2
c2
34 b_tmp = 0x7FFF&b
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

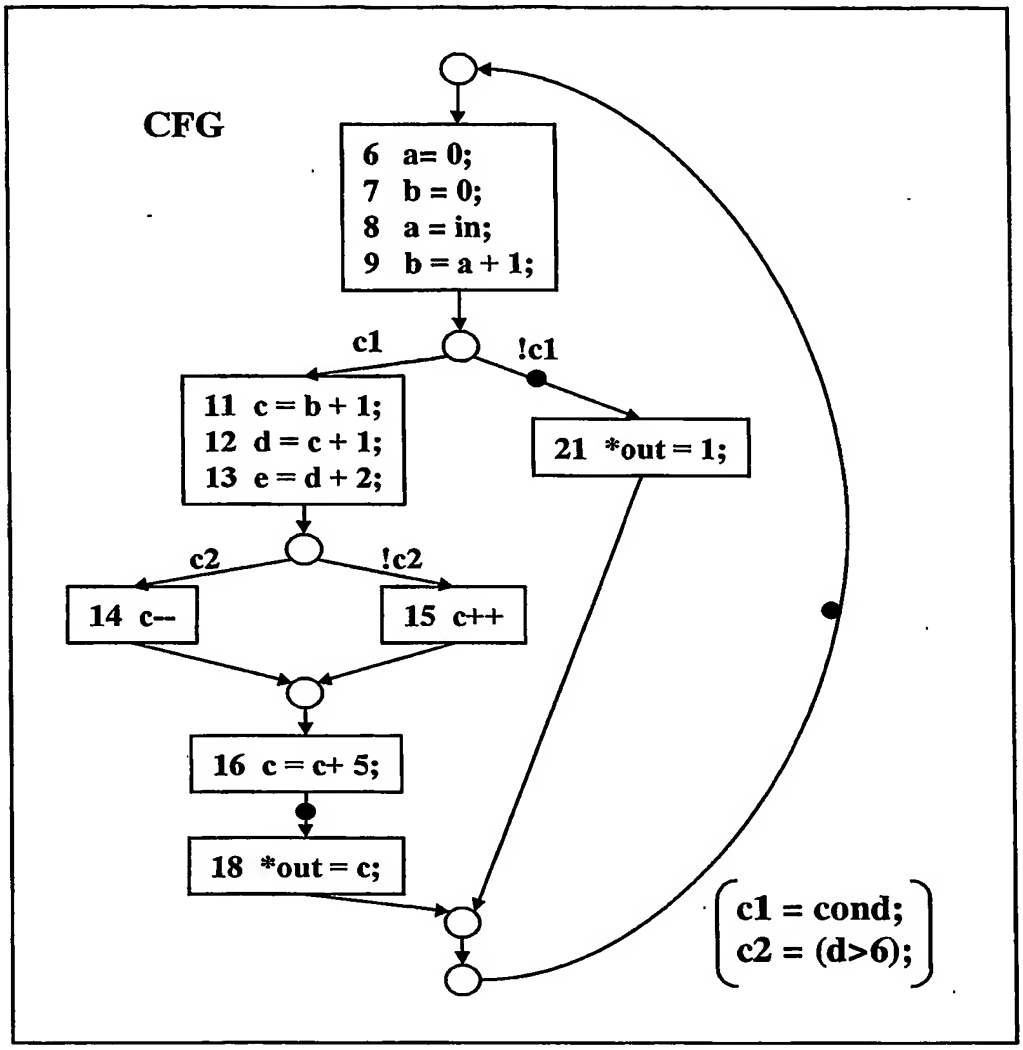
第３１図


ステート割り当て
ST0
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
~c1
ST2
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
c1
31 a_tmp = 0x7FFF&a;
ST1
~c2
c2
34 b_tmp = 0x7FFF&b
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

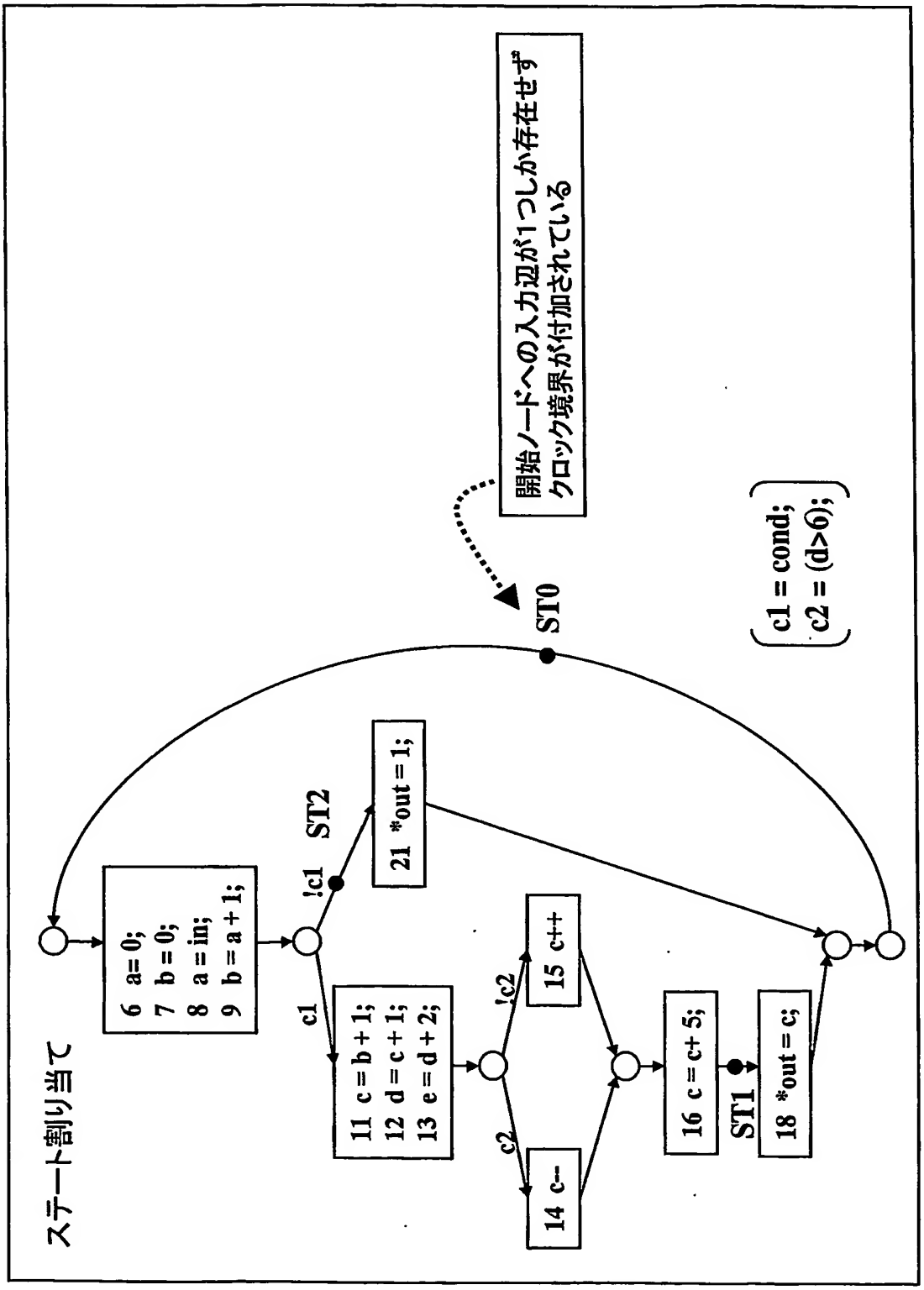
第３２図

```
1  void foo(unsigned short in,
2               unsigned short cond,
3               unsigned short *out) {
4    unsigned short a, b, c, d, e;
5    while(1) {
6      a= 0;
7      b = 0;
8      a = in;
9      b = a + 1;
10     if (cond) {
11       c = b + 1;
12       d = c + 1;
13       e  = d + 2;
14       if (d > 6) c--;
15       else       c++;
16       c = c + 5;
17       $
18       *out = c;
19     } else {
20       $
21       *out = 1;
22     }
23     $
24 }
```

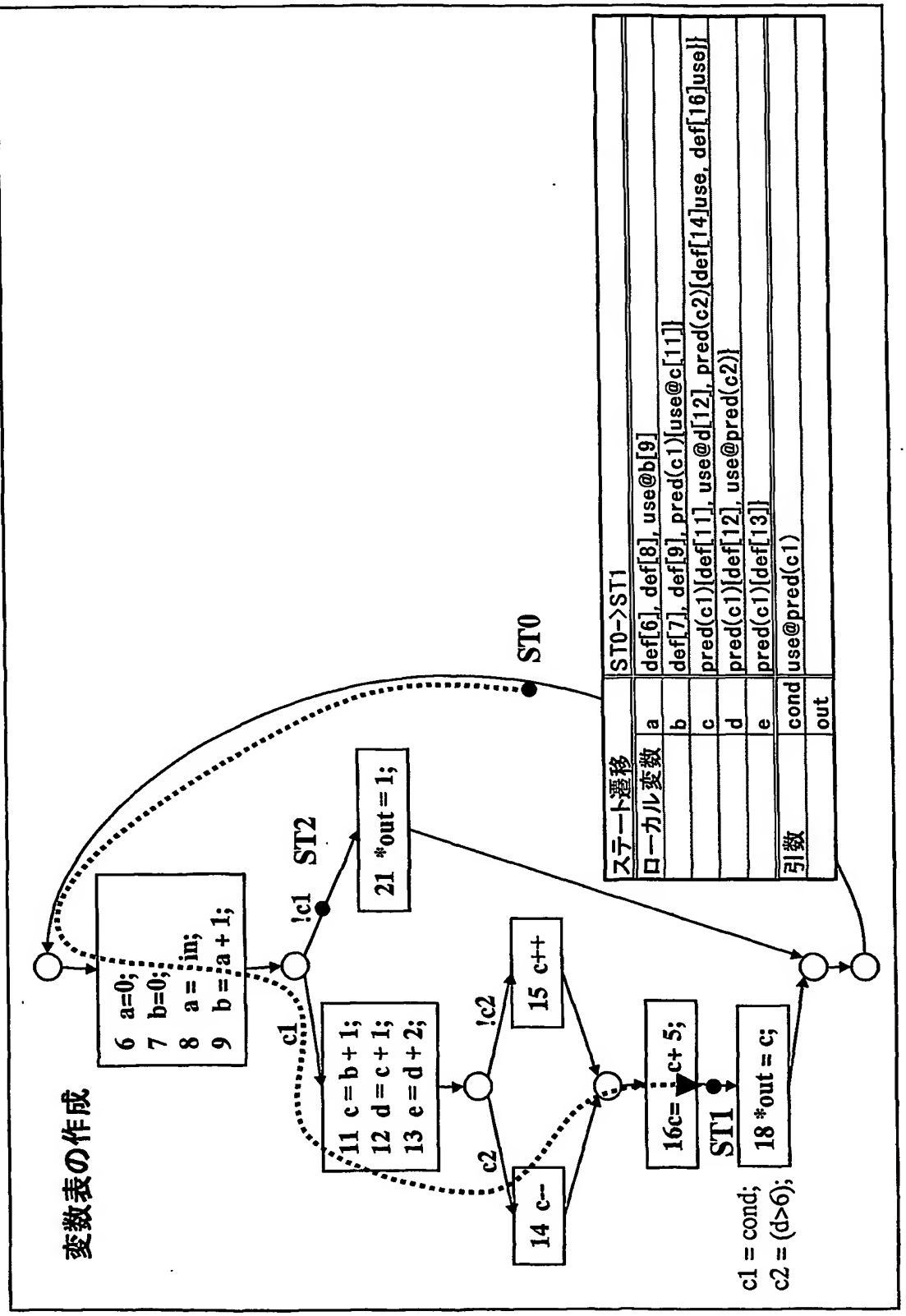
第３３図


CFG
6 a= 0;
7 b = 0;
8 a = in;
9 b = a + 1;
c1
!c1
11 c = b + 1;
12 d = c + 1;
13 e = d + 2;
21 *out = 1;
c2
!c2
14 c--
15 c++
16 c = c+ 5;
18 *out = c;
c1 = cond;
c2 = (d>6);

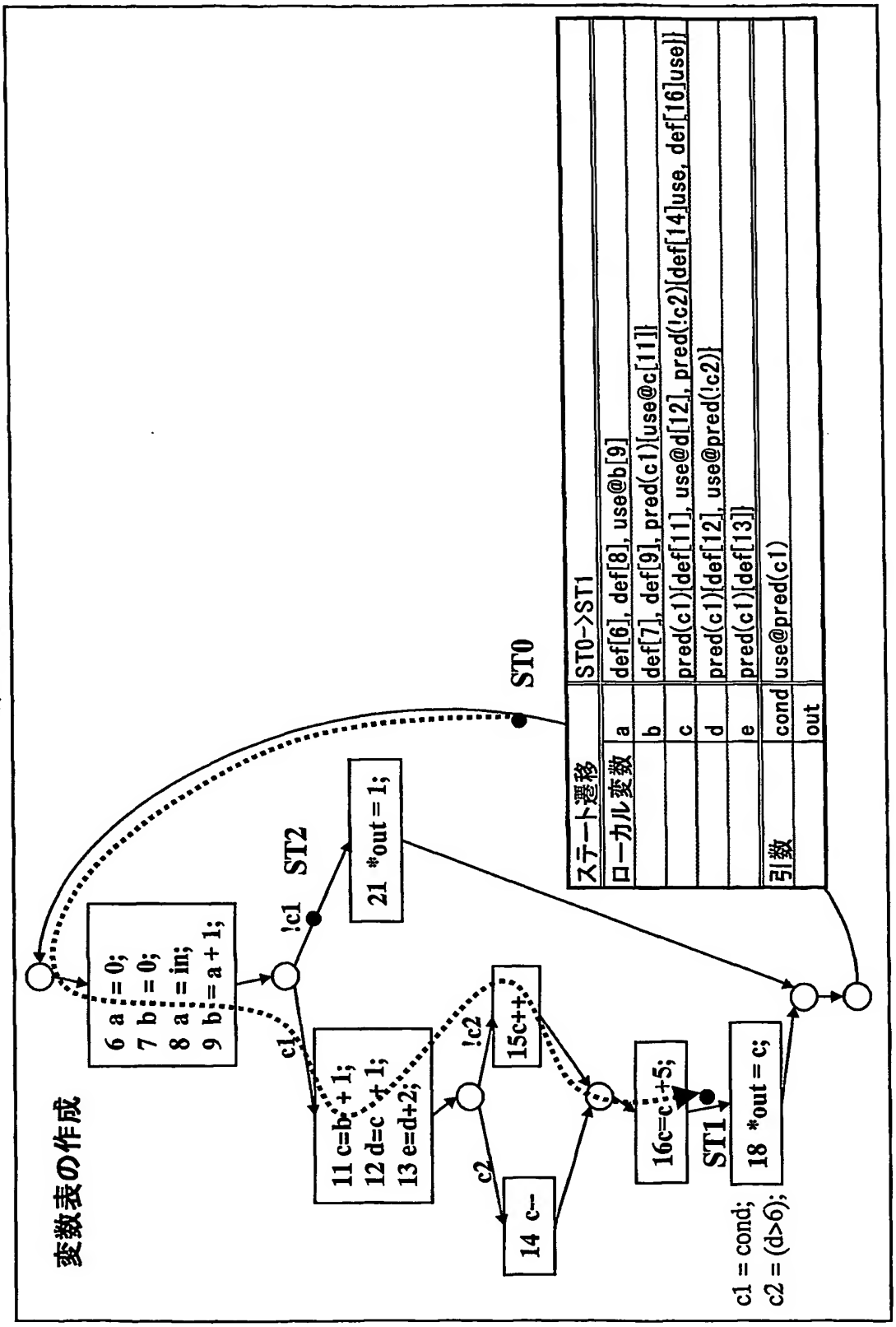
第３４図


ステート割り当て
6 a= 0;
7 b = 0;
8 a = in;
9 b = a + 1;
!c1
ST2
21 *out = 1;
c1
11 c = b + 1;
12 d = c + 1;
13 e = d + 2;
!c2
15 c++
c2
14 c--
16 c = c+ 5;
ST1
18 *out = c;
ST0
開始ノードへの入力辺が1つしか存在せず
クロック境界が付加されている
c1 = cond;
c2 = (d>6);

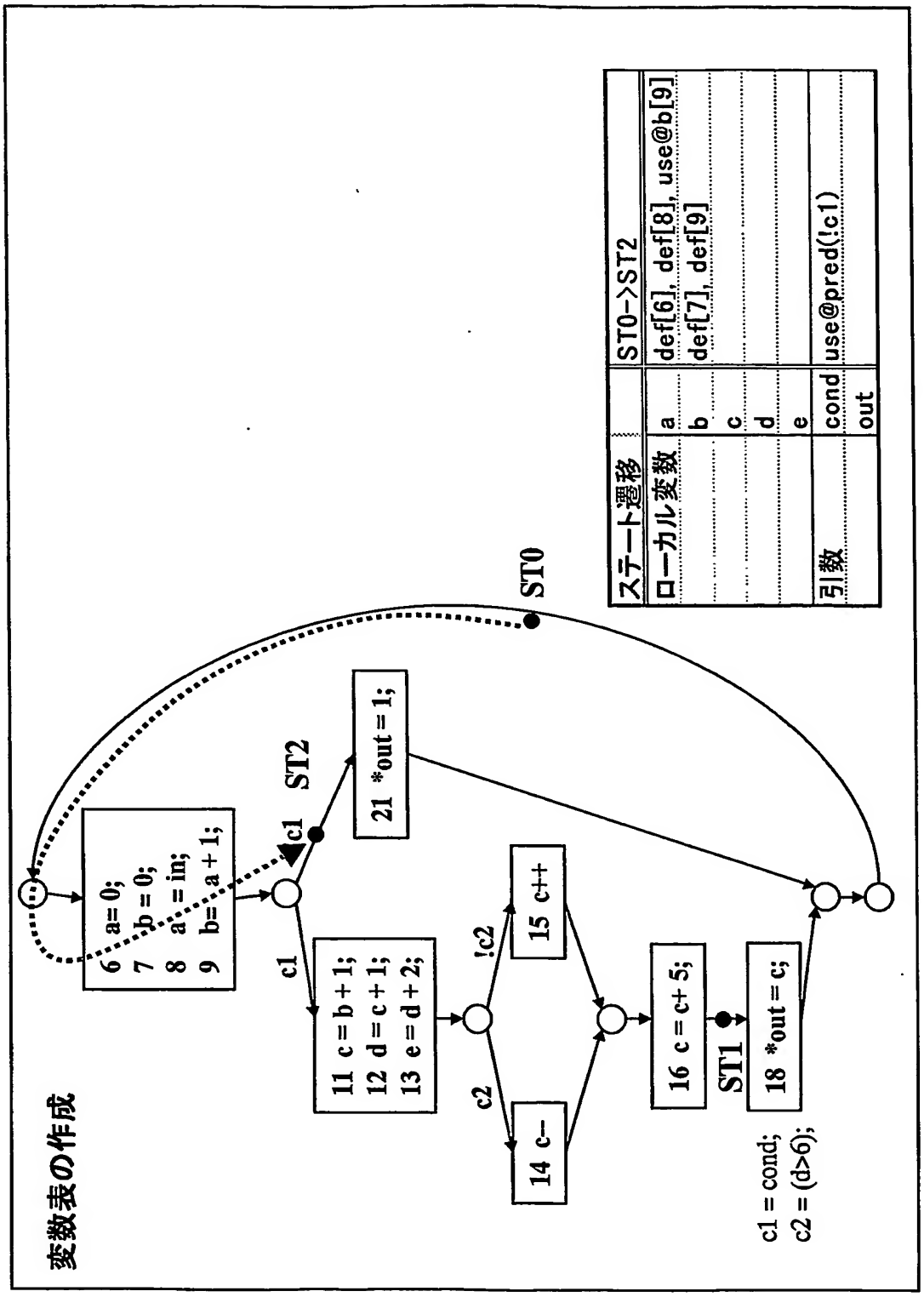
第35図


変数表の作成
6 a=0;
7 b=0;
8 a = in;
9 b = a + 1;
ST0
ST0->ST1
ステート遷移
ローカル変数
a
b
c
d
e
引数
cond
out
def[6], def[8], use@b[9]
def[7], def[9], pred(c1)[use@c[11]]
pred(c1)[def[11], use@d[12], pred(c2)[def[14]use, def[16]use]]
pred(c1)[def[12], use@pred(c2)]
pred(c1)[def[13]]
use@pred(c1)
!c1
ST2
21 *out = 1;
c1
11 c = b + 1;
12 d = c + 1;
13 e = d + 2;
!c2
15 c++
c2
14 c--
16c = c+ 5;
ST1
18 *out = c;
c1 = cond;
c2 = (d>6);

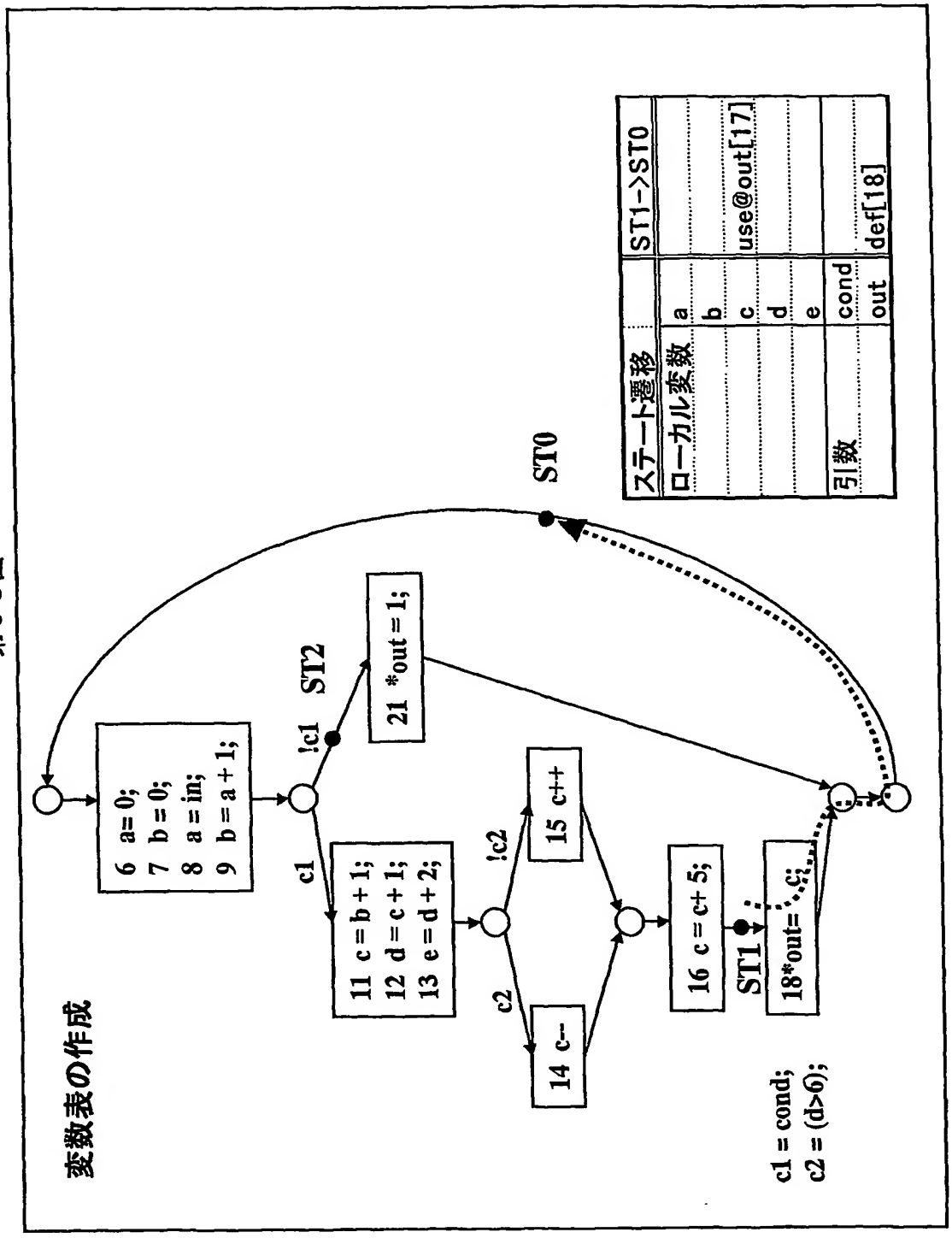
第36図


変数表の作成
6 a = 0;
7 b = 0;
8 a = in;
9 b = a + 1;
c1
!c1
ST2
11 c=b + 1;
12 d=c + 1;
13 e=d+2;
21 *out = 1;
c2
!c2
14 c--
15c++;
16c=c++5;
ST1
18 *out = c;
c1 = cond;
c2 = (d>6);
ST0
ステート遷移
ST0->ST1
ローカル変数
a
def[6], def[8], use@b[9]
b
def[7], def[9], pred(c1)[use@c[11]]
c
pred(c1)[def[11], use@d[12], pred(!c2)[def[14]use, def[16]use]]
d
pred(c1)[def[12], use@pred(!c2)]
e
pred(c1)[def[13]]
引数
cond
use@pred(c1)
out

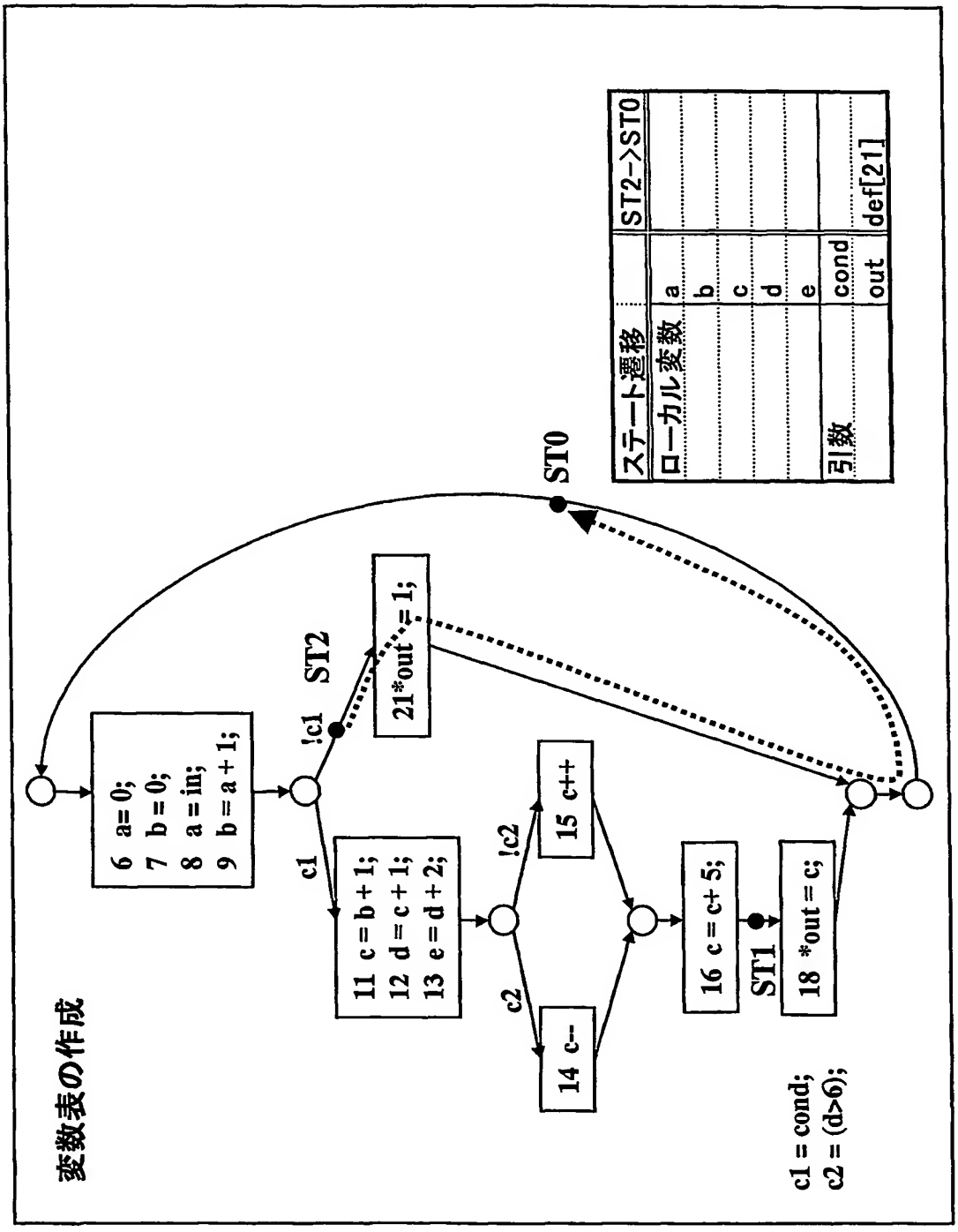
第37図


変数表の作成
ST0
ST1
ST2
c1
c2
!c2
6 a= 0;
7 b = 0;
8 a = in;
9 b= a + 1;
11 c = b + 1;
12 d = c + 1;
13 e = d + 2;
14 c--
15 c++
16 c = c+ 5;
18 *out = c;
21 *out = 1;
c1 = cond;
c2 = (d>6);
ステート遷移
ST0->ST2
ローカル変数
a
b
c
d
e
def[6], def[8], use@b[9]
def[7], def[9]
引数
cond
out
use@pred(!c1)

第３８図


変数表の作成
6 a= 0;
7 b = 0;
8 a = in;
9 b = a + 1;
!c1
ST2
21 *out = 1;
c1
11 c = b + 1;
12 d = c + 1;
13 e = d + 2;
!c2
15 c++
c2
14 c--
16 c = c+ 5;
ST1
18*out= c;
c1 = cond;
c2 = (d>6);
ST0
ステート遷移
ST1->ST0
ローカル変数
a
b
c
use@out[17]
d
e
引数
cond
out
def[18]

第39図


変数表の作成
6 a= 0;
7 b= 0;
8 a= in;
9 b= a+ 1;
ST0
!c1 ST2
21*out += 1;
c1
11 c= b+ 1;
12 d= c+ 1;
13 e= d+ 2;
!c2
15 c++
c2
14 c--
16 c= c+ 5;
ST1
18 *out= c;
c1 = cond;
c2 = (d>6);
ステート遷移
ST2->ST0
ローカル変数
a
b
c
d
e
引数
cond
out
def[21]

第４０図

変数表作成

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[6], def[8], use@b[9]	def[6], def[8], use@b[9]		
	b	def[7], def[9], pred(c1){use@c[11]}	def[7], def[9]		
	c	pred(c1){def[11], use@d[12], pred(c2){def[14]use, def[16]use}}		use@out[17]	
	d	pred(c1){def[12], use@pred(c2)}			
	e	pred(c1){def[13]}			
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]

ステート遷移		ST0->ST1
ローカル変数	a	def[6], def[8], use@b[9]
	b	def[7], def[9], pred(c1){use@c[11]}
	c	pred(c1){def[11], use@d[12], pred(!c2){def[14]use, def[16]use}}
	d	pred(c1){def[12], use@pred(!c2)}
	e	pred(c1){def[13]}
引数	cond	use@pred(c1)
	out	

def[n] ：n行目で変数定義されている事を表す
use@var[m] ：m行目で変数varへの代入に用いられている事を表す
pred(cond){...} ：条件condの分岐が成立した場合、{...}が実施される事を表す
def[l]use ：l行目で自変数への代入に用いられている事を表す
use@pred(cond) ：条件condで用いられている事を表す

第４１図

冗長ステートメント削除

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[6], def[8], use@b[9]	def[6], def[8], use@b[9]		
	b	def[7], def[9], pred(c1){use@c[11]}	def[7], def[9]		
	c	pred(c1){def[11], use@d[12], pred(c2){def[14]use, def[16]use}}		use@out[17]	
	d	pred(c1){def[12], use@pred(c2)}			
	e	pred(c1){def[13]}			
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]

ステート遷移		ST0->ST1
ローカル変数	a	def[6], def[8], use@b[9]
	b	def[7], def[9], pred(c1){use@c[11]}
	c	pred(c1){def[11], use@d[12], pred(!c2){def[14]use, def[16]use}}
	d	pred(c1){def[12], use@pred(!c2)}
	e	pred(c1){def[13]}
引数	cond	use@pred(c1)
	out	

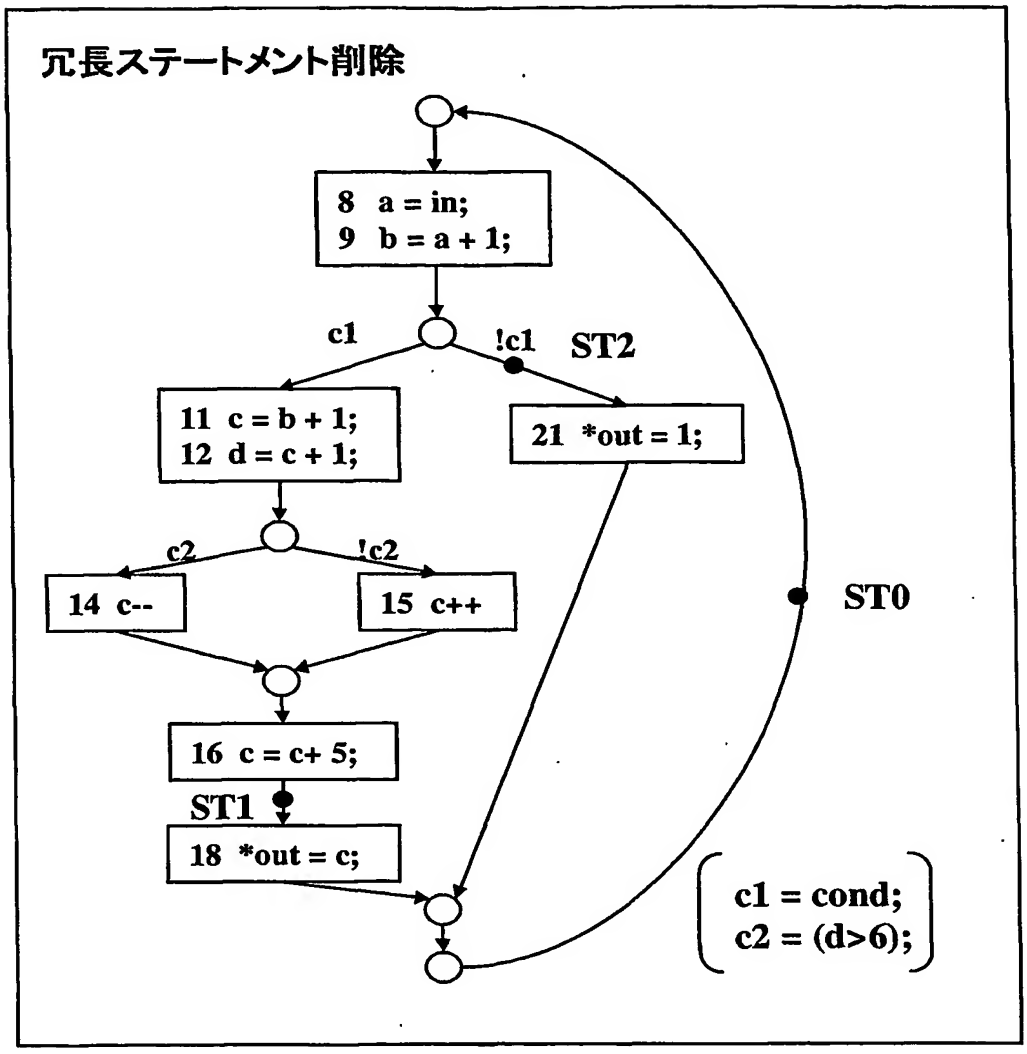
差替え用紙（規則26）

第42図

冗長ステートメント削除

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[8], use@b[9]	def[8], use@b[9]		
	b	def[9], pred(c1){use@c[11]}	def[9]		
	c	pred(c1){def[11], use@d[12], pred(c2){def[14]use, def[16]use}}		use@out[17]	
	d	pred(c1){def[12], use@pred(c2)}			
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]
ステート遷移		ST0->ST1			
ローカル変数	a	def[8], use@b[9]			
	b	def[9], pred(c1){use@c[11]}			
	c	pred(c1){def[11], use@d[12], pred(!c2){def[14]use, def[16]use}}			
	d	pred(c1){def[12], use@pred(!c2)}			
引数	cond	use@pred(c1)			
	out				

第４３図

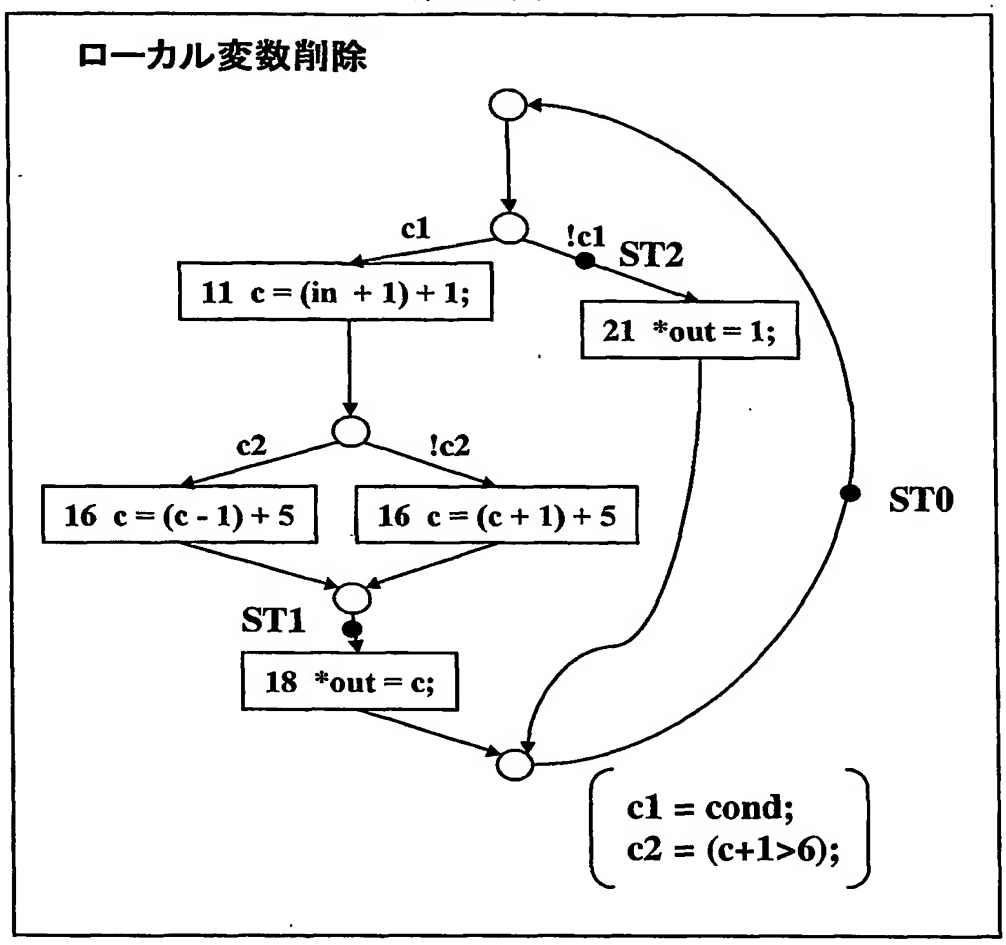

冗長ステートメント削除
8 a = in;
9 b = a + 1;
c1
!c1
ST2
11 c = b + 1;
12 d = c + 1;
21 *out = 1;
c2
!c2
14 c--
15 c++
ST0
16 c = c+ 5;
ST1
18 *out = c;
c1 = cond;
c2 = (d>6);

第４４図

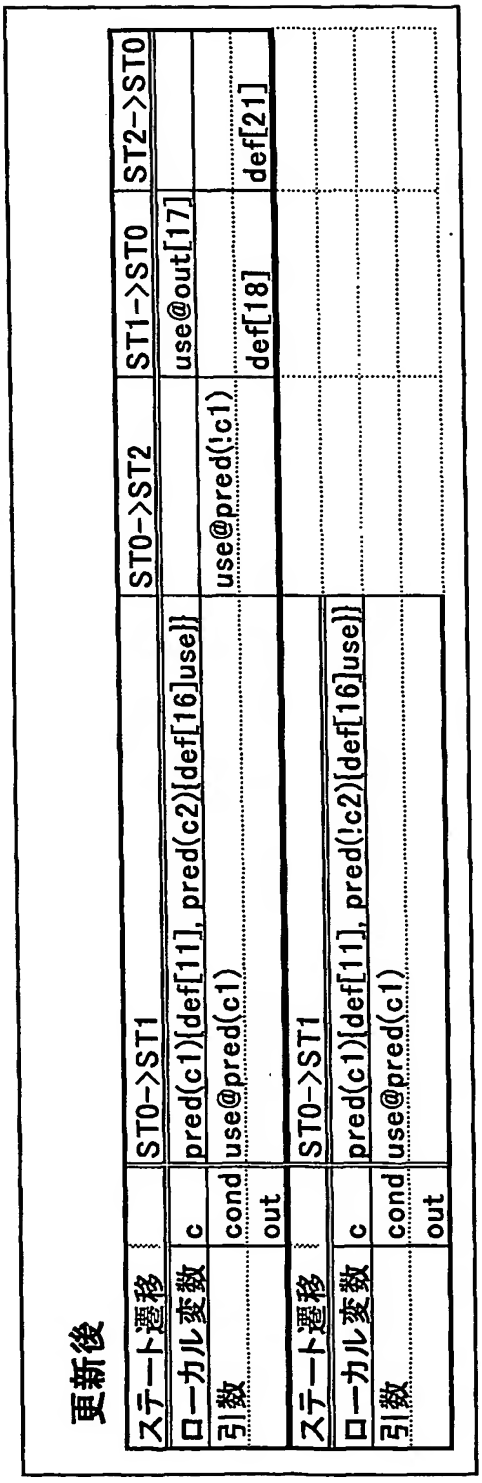
ローカル変数削除

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[8], use@b[9]	def[8], use@b[9]		
	b	def[9], pred(c1)|use@c[11]]	def[9]		
	c	pred(c1)|def[11], use@d[12], pred(c2)|def[14]use, def[16]use]]		use@out[17]	
	d	pred(c1)|def[12], use@pred(c2)]			
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]
ステート遷移		ST0->ST1			
ローカル変数	a	def[8], use@b[9]			
	b	def[9], pred(c1)|use@c[11]]			
	c	pred(c1)|def[11], use@d[12], pred(!c2)|def[14]use, def[16]use]]			
	d	pred(c1)|def[12], use@pred(!c2)]			
引数	cond	use@pred(c1)			
	out				

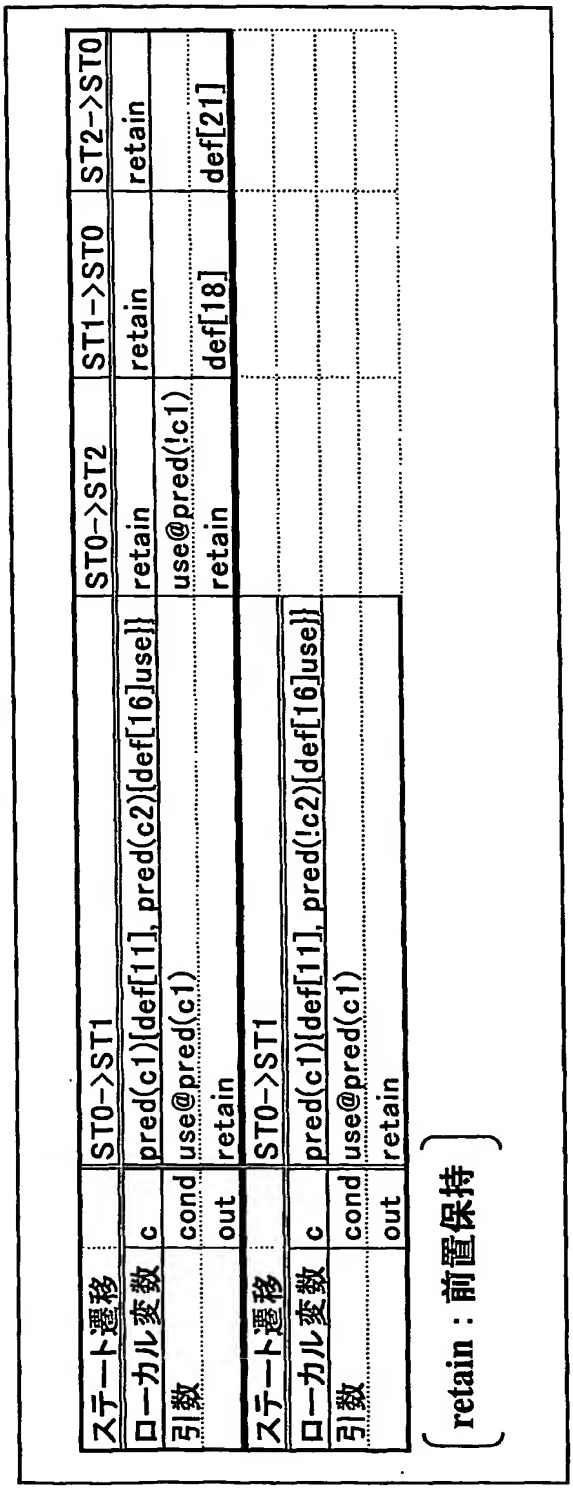
第45図


ローカル変数削除
c1
!c1
ST2
11 c = (in + 1) + 1;
21 *out = 1;
c2
!c2
16 c = (c - 1) + 5
16 c = (c + 1) + 5
ST0
ST1
18 *out = c;
c1 = cond;
c2 = (c+1>6);

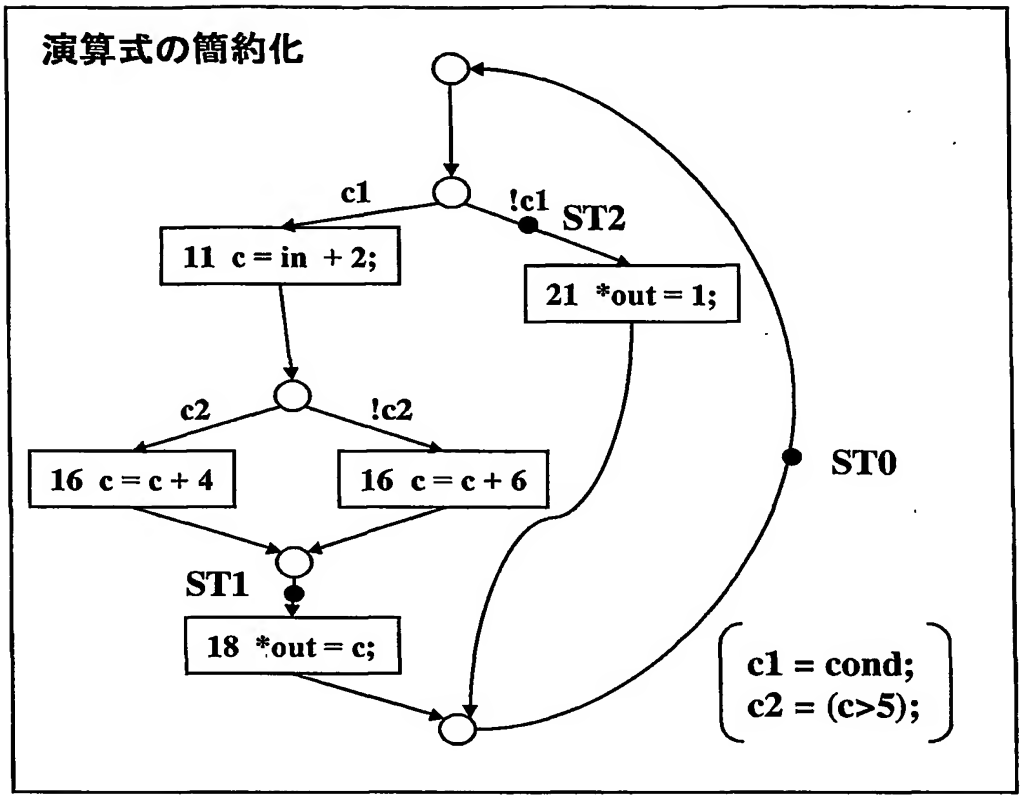
第46図


更新後
ステート遷移
ST0->ST1
ST0->ST2
ST1->ST0
ST2->ST0
ローカル変数
c
pred(c1){def[11], pred(c2){def[16]use}}
use@out[17]
引数
cond
use@pred(c1)
use@pred(!c1)
def[18]
def[21]
out
ステート遷移
ST0->ST1
ローカル変数
c
pred(c1){def[11], pred(!c2){def[16]use}}
引数
cond
use@pred(c1)
out

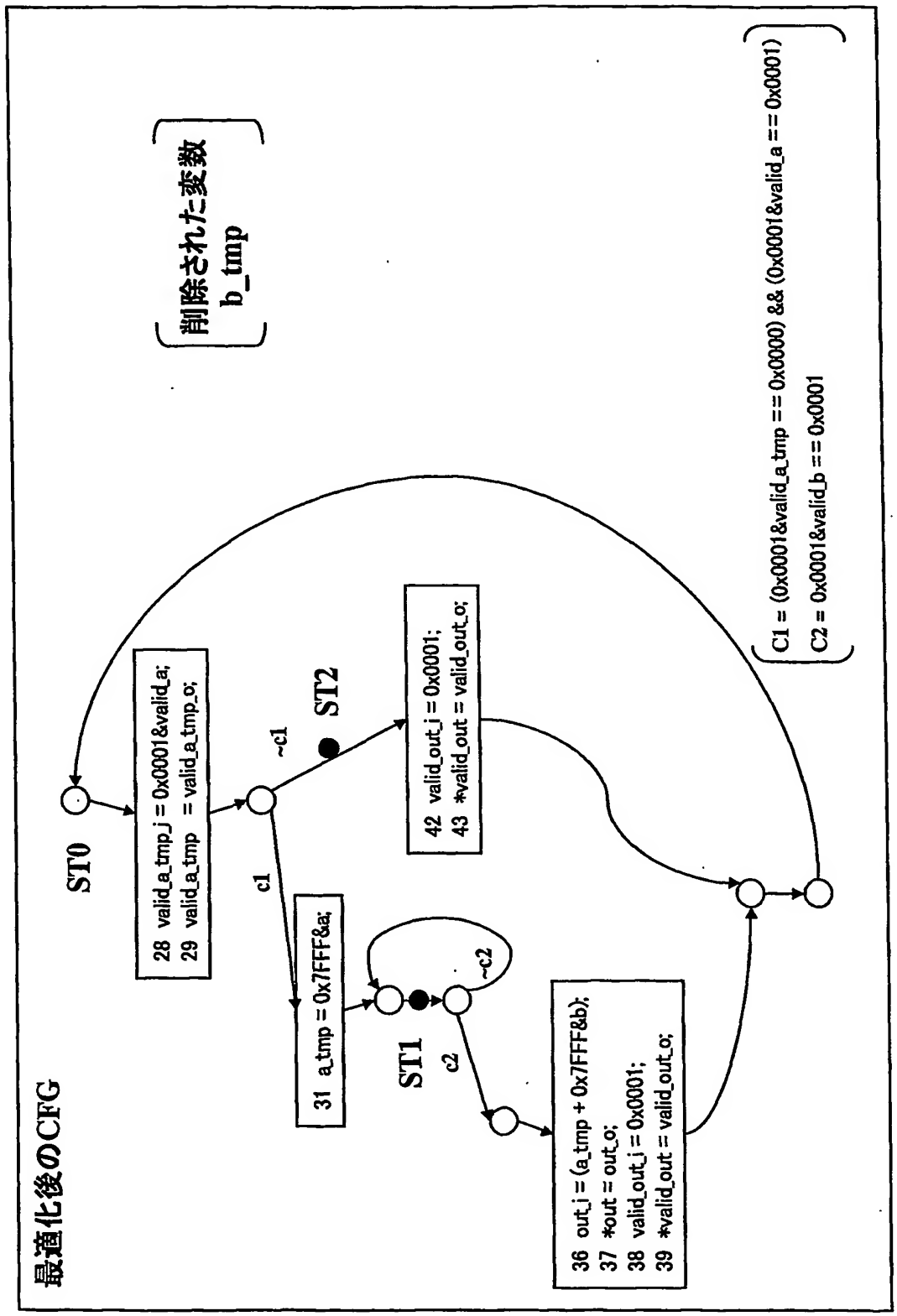
第47図


ステート遷移
ST0->ST1
ST0->ST2
ST1->ST0
ST2->ST0
ローカル変数
c
pred(c1){def[11], pred(c2){def[16]use}}
retain
retain
retain
引数
cond
use@pred(c1)
use@pred(!c1)
out
retain
def[18]
def[21]
ステート遷移
ST0->ST1
ローカル変数
c
pred(c1){def[11], pred(!c2){def[16]use}}
引数
cond
use@pred(c1)
out
retain
retain：前置保持

第４８図


演算式の簡約化
c1
!c1
ST2
11 c = in + 2;
21 *out = 1;
c2
!c2
16 c = c + 4
16 c = c + 6
ST0
ST1
18 *out = c;
c1 = cond;
c2 = (c>5);

第４９図


最適化後のCFG
削除された変数
b_tmp
ST0
ST1
ST2
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
~c1
31 a_tmp = 0x7FFF&a;
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
c2
~c2
36 out_j = (a_tmp + 0x7FFF&b);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

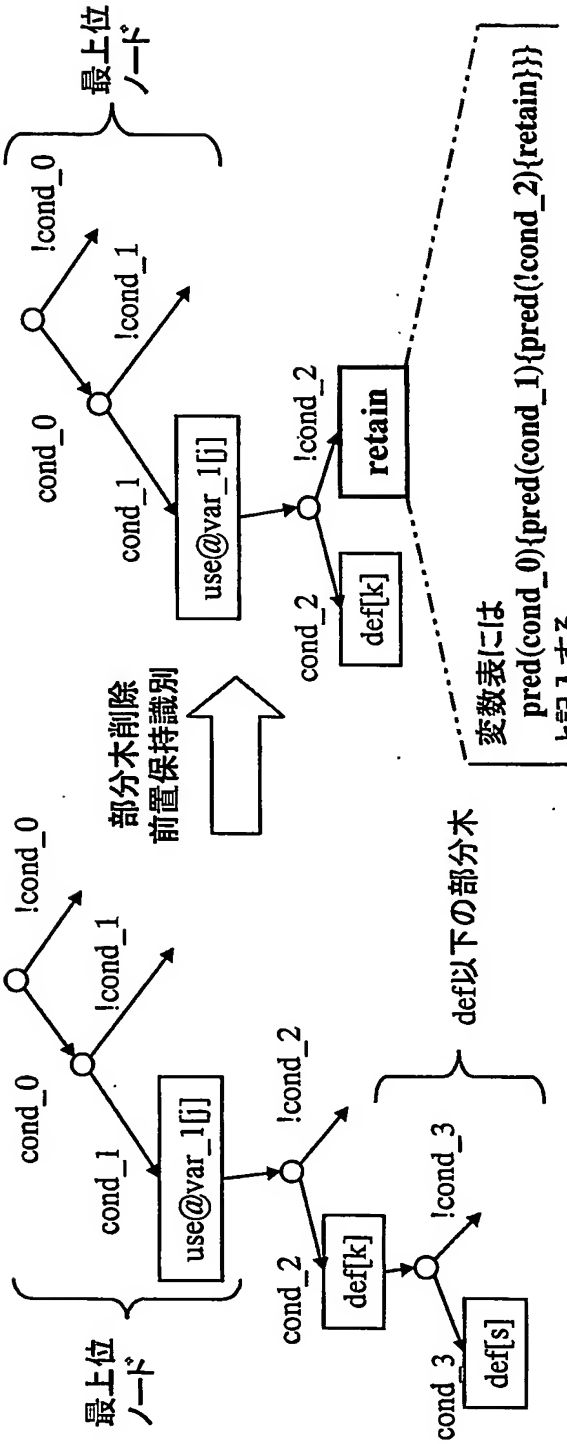
第５０図

最適化後

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST1
ローカル変数	valid_a_tmp	def[29], use@pred(c1)	def[29], use@pred(!c1)	pred(c2){def[29], use@pred(c1)}
	valid_a_tmp_I	def[28]	def[28]	pred(c2){def[28]}
	a_tmp	pred(c1){def[31]}		pred(c2){pred(c1){def[31]}}
	out_I			pred(c2){def[36]}
	valid_out_i			pred(c2){def[38]}
引数	a	pred(c1){use@a_tmp[31]}		pred(c2){pred(c1){use@a_tmp[31]}}
	b			pred(c2){use@out_i[36]}
	valid_a	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(!c1)	pred(c2){use@valid_a_tmp_i[28], use@pred(c1)}
	valid_b			use@pred(c2)
	valid_out			pred(c2){def[39]}
	out			pred(c2){def[37]}
	valid_a_tmp_o	use@valid_a_tmp[29]	use@valid_a_tmp[29]	pred(c2){use@valid_a_tmp[29]}
	valid_out_o			pred(c2){use@valid_out_o[39]}
	out_o			pred(c2){use@out[37]}

ステート遷移		ST1->ST2	ST2->ST1	ST2->ST2
ローカル変数	valid_a_tmp	pred(c2){def[29], use@pred(!c1)}	def[29], use@pred(c1)	def[29], use@pred(!c1)
	valid_a_tmp_I	pred(c2){def[28]}	def[28]	def[28]
	a_tmp		pred(c1){def[31]}	
	out_I	pred(c2){def[36]}		
	valid_out_i	pred(c2){def[38]}	def[42]	def[42]
引数	a		pred(c1){use@a_tmp[31]}	
	b	pred(c2){use@out_i[36]}		
	valid_a	pred(c2){use@valid_a_tmp_i[28], use@pred(c1)}	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(!c1)
	valid_b	use@pred(c2)		
	valid_out	pred(c2){def[39]}	def[43]	def[43]
	out	pred(c2){def[37]}		
	valid_a_tmp_o	pred(c2){use@valid_a_tmp[29]}		
	valid_out_o	pred(c2){use@valid_out_o[39]}	use@valid_out[43]	use@valid_out[43]
	out_o	pred(c2){use@out[37]}		

第５１図


最上位ノード
cond_0
!cond_0
cond_1
!cond_1
use@var_1[j]
cond_2
!cond_2
def[k]
cond_3
!cond_3
def[s]
retain
部分木削除
前置保持識別
def以下の部分木
変数表には
pred(cond_0){pred(cond_1){pred(!cond_2){retain}}}
と記入する

第５２図

前置保持解析実行後

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST1	ST1->ST1
ローカル変数	valid_a_tmp	def[29], use@pred(c1)	def[29], use@pred(!c1)	pred(c2){def[29], use@pred(c1)}	pred(!c2){retain}
	valid_a_tmp_I	def[28]	def[28]	pred(c2){def[28]}	pred(!c2){retain}
	a_tmp	pred(c1){def[31]}	pred(!c1){retain}	pred(c2){pred(c1){def[31]}}	pred(!c2){retain}
	out_I	retain	retain	pred(c2){def[36]}	pred(!c2){retain}
	valid_out_i	retain	retain	pred(c2){def[38]}	pred(!c2){retain}
引数	a	pred(c1){use@a_tmp[31]}		pred(c2){pred(c1){use@a_tmp[31]}}	
	b			pred(c2){use@out_i[36]}	
	valid_a	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(!c1)	pred(c2){use@valid_a_tmp_i[28], use@pred(c1)}	
	valid_b			use@pred(c2)	use@pred(!c2)
	valid_out	retain	retain	pred(c2){def[39]}	pred(!c2){retain}
	out	retain	retain	pred(c2){def[37]}	pred(!c2){retain}
	valid_a_tmp_o	use@valid_a_tmp[29]	use@valid_a_tmp[29]	pred(c2){use@valid_a_tmp[29]}	
	valid_out_o			pred(c2){use@valid_out_o[39]}	
	out_o			pred(c2){use@out[37]}	

ステート遷移		ST1->ST2	ST2->ST1	ST2->ST2
ローカル変数	valid_a_tmp	pred(c2){def[29], use@pred(!c1)}	def[29], use@pred(c1)	def[29], use@pred(!c1)
	valid_a_tmp_I	pred(c2){def[28]}	def[28]	def[28]
	a_tmp	pred(!c1){retain}	pred(c1){def[31]}	pred(!c1){retain}
	out_I	pred(c2){def[36]}	retain	retain
	valid_out_i	pred(c2){def[38]}	def[42]	def[42]
引数	a		pred(c1){use@a_tmp[31]}	
	b	pred(c2){use@out_i[36]}		
	valid_a	pred(c2){use@valid_a_tmp_i[28], use@pred(c1)}	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(!c1)
	valid_b	use@pred(c2)		
	valid_out	pred(c2){def[39]}	def[43]	def[43]
	out	pred(c2){def[37]}	retain	retain
	valid_a_tmp_o	pred(c2){use@valid_a_tmp[29]}		
	valid_out_o	pred(c2){use@valid_out_o[39]}	use@valid_out[43]	use@valid_out[43]
	out_o	pred(c2){use@out[37]}		

第５２図

前置保持解析実行後

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST1	ST1->ST1
ローカル変数	valid_a_tmp	def[29], use@pred(c1)	def[29], use@pred(c1)	pred(c2)[def[29], use@pred(c1)]	pred(!c2)[retain]
	valid_a_tmp_i	def[28]	def[28]	pred(c2)[def[28]]	pred(!c2)[retain]
	a_tmp	pred(c1)[def[31]]	pred(c1)[retain]	pred(c2)[pred(c1)[def[31]]]	pred(!c2)[retain]
	out_i	retain	retain	pred(c2)[def[36]]	pred(!c2)[retain]
	valid_out_i	retain	retain	pred(c2)[def[38]]	pred(!c2)[retain]
引数	a	pred(c1)[use@a_tmp[31]]		pred(c2)[pred(c1)[use@a_tmp[31]]]	
	b			pred(c2)[use@out_i[36]]	
	valid_a	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(c1)	pred(c2)[use@valid_a_tmp_i[28], use@pred(c1)]	
	valid_b			use@pred(c2)	use@pred(!c2)
	valid_out	retain	retain	pred(c2)[def[39]]	pred(!c2)[retain]
	out	retain	retain	pred(c2)[def[37]]	pred(!c2)[retain]
	valid_a_tmp_o	use@valid_a_tmp[29]	use@valid_a_tmp[29]	pred(c2)[use@valid_a_tmp[29]]	
	valid_out_o			pred(c2)[use@valid_out_o[39]]	
	out_o			pred(c2)[use@out[37]]	
ステート遷移		ST1->ST2	ST2->ST1	ST2->ST2	
ローカル変数	valid_a_tmp	pred(c2)[def[29], use@pred(!c1)]	def[29], use@pred(c1)	def[29], use@pred(!c1)	
	valid_a_tmp_i	pred(c2)[def[28]]	def[28]	def[28]	
	a_tmp	pred(!c1)[retain]	pred(c1)[def[31]]	pred(!c1)[retain]	
	out_i	pred(c2)[def[36]]	retain	retain	
	valid_out_i	pred(c2)[def[38]]	def[42]	def[42]	
引数	a		pred(c1)[use@a_tmp[31]]		
	b	pred(c2)[use@out_i[36]]			
	valid_a	pred(c2)[use@valid_a_tmp_i[28], use@pred(c1)]	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(!c1)	
	valid_b	use@pred(c2)			
	valid_out	pred(c2)[def[39]]	def[43]	def[43]	
	out	pred(c2)[def[37]]	retain	retain	
	valid_a_tmp_o	pred(c2)[use@valid_a_tmp[29]]			
	valid_out_o	pred(c2)[use@valid_out_o[39]]	use@valid_out[43]	use@valid_out[43]	
	out_o	pred(c2)[use@out[37]]			

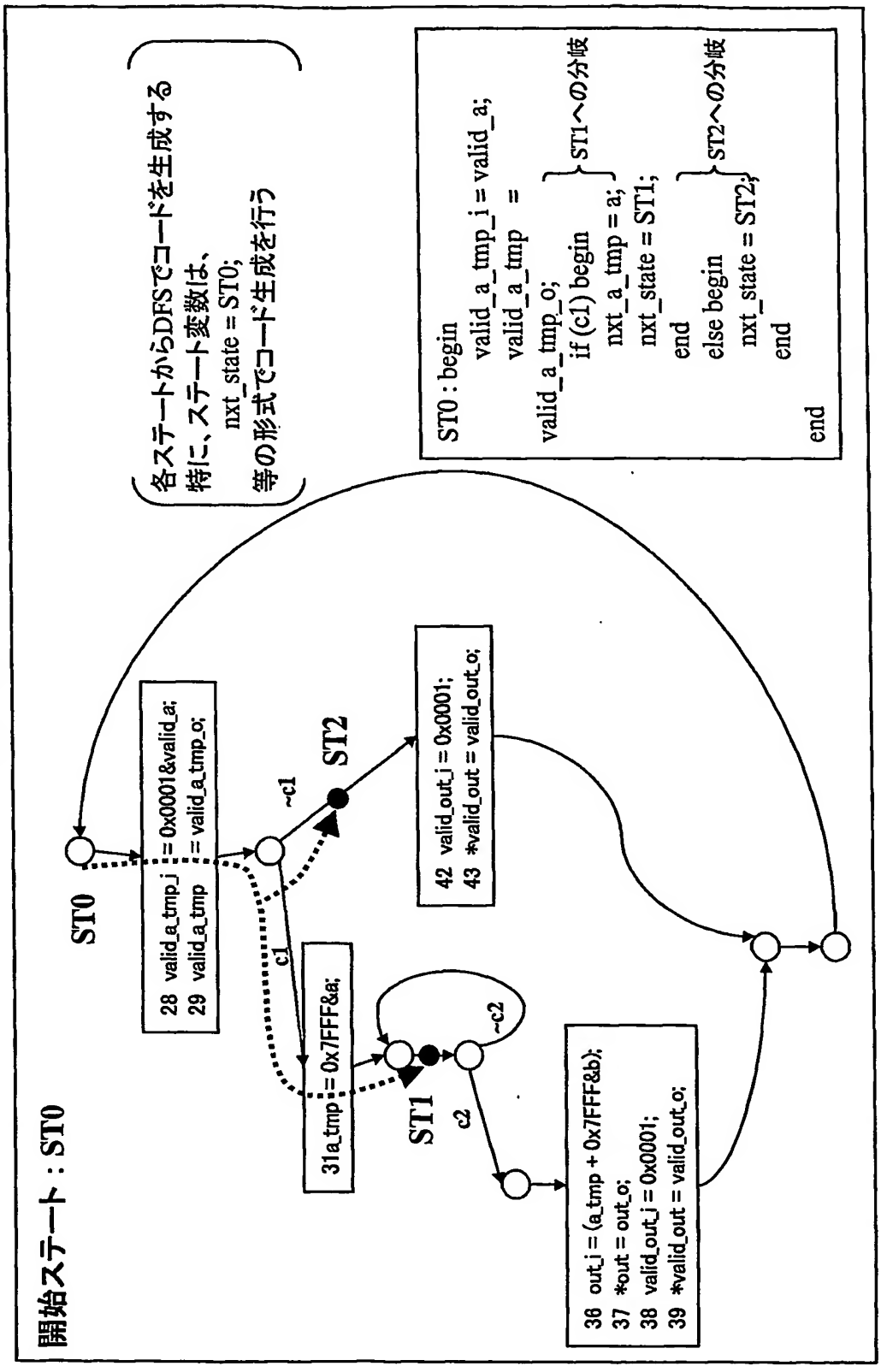
第５３図

前置保持解析結果の変数表からの情報取得

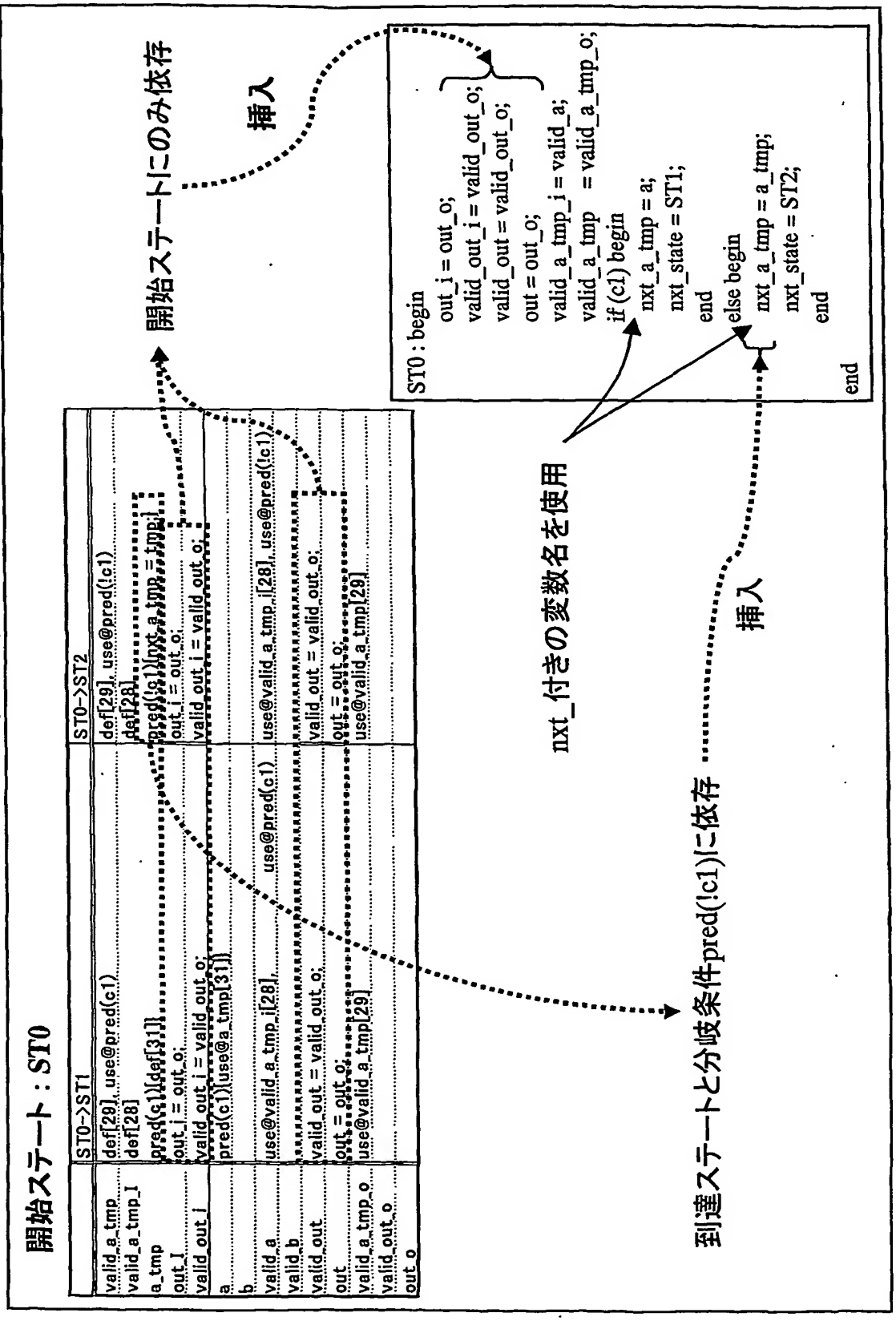
ステート遷移		ST0->ST1	ST0->ST2	ST1->ST1	ST1->ST1
ローカル変数	valid_a_tmp	def[29], use@pred(c1)	def[29], use@pred(!c1)	pred(c2){def[29], use@pred(c1)}	pred(!c2){valid_a_tmp = valid_a_tmp_o;}
	valid_a_tmp_i	def[28]	def[28]	pred(c2){def[28]}	pred(!c2){valid_a_tmp_i = valid_a_tmp_o;}
	a_tmp	pred(c1){def[31]}	pred(!c1){nxt_a_tmp = tmp;}	pred(c2){pred(c1){def[31]}}	pred(!c2){nxt_a_tmp = a_tmp;}
	out_i	out_i = out_o;	out_i = out_o;	pred(c2){def[36]}	pred(!c2){out_i = out_o;}
	valid_out_i	valid_out_i = valid_out_o;	valid_out_i = valid_out_o;	pred(c2){def[38]}	pred(!c2){valid_out_i = valid_out_o;}
引数	a	pred(c1){use@a_tmp[31]}		pred(c2){pred(c1){use@a_tmp[31]}}	
	b			pred(c2){use@out_i[36]}	
	valid_a	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(!c1)	pred(c2){use@valid_a_tmp_i[28], use@pred(c1)}	
	valid_b			use@pred(c2)	use@pred(!c2)
	valid_out	valid_out = valid_out_o;	valid_out = valid_out_o;	pred(c2){def[39]}	pred(!c2){valid_out = valid_out_o;}
	out	out = out_o;	out = out_o;	pred(c2){def[37]}	pred(!c2){out = out_o;}
	valid_a_tmp_o	use@valid_a_tmp[29]	use@valid_a_tmp[29]	pred(c2){use@valid_a_tmp[29]}	
	valid_out_o			pred(c2){use@valid_out_o[39]}	
	out_o			pred(c2){use@out[37]}	

ステート遷移		ST1->ST2	ST2->ST1	ST2->ST2
ローカル変数	valid_a_tmp	pred(c2){def[29], use@pred(!c1)}	def[29], use@pred(c1)	def[29], use@pred(!c1)
	valid_a_tmp_i	pred(c2){def[28]}	def[28]	def[28]
	a_tmp	pred(!c1){nxt_a_tmp = a_tmp;}	pred(c1){def[31]}	pred(!c1){nxt_a_tmp = a_tmp;}
	out_i	pred(c2){def[36]}	out_i = out_o;	out_i = out_o;
	valid_out_i	pred(c2){def[38]}	def[42]	def[42]
引数	a		pred(c1){use@a_tmp[31]}	
	b	pred(c2){use@out_i[36]}		
	valid_a	pred(c2){use@valid_a_tmp_i[28], use@pred(c1)}	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(!c1)
	valid_b	use@pred(c2)		
	valid_out	pred(c2){def[39]}	def[43]	def[43]
	out	pred(c2){def[37]}	out = out_o;	out = out_o;
	valid_a_tmp_o	pred(c2){use@valid_a_tmp[29]}		
	valid_out_o	pred(c2){use@valid_out_o[39]}	use@valid_out[43]	use@valid_out[43]
	out_o	pred(c2){use@out[37]}		

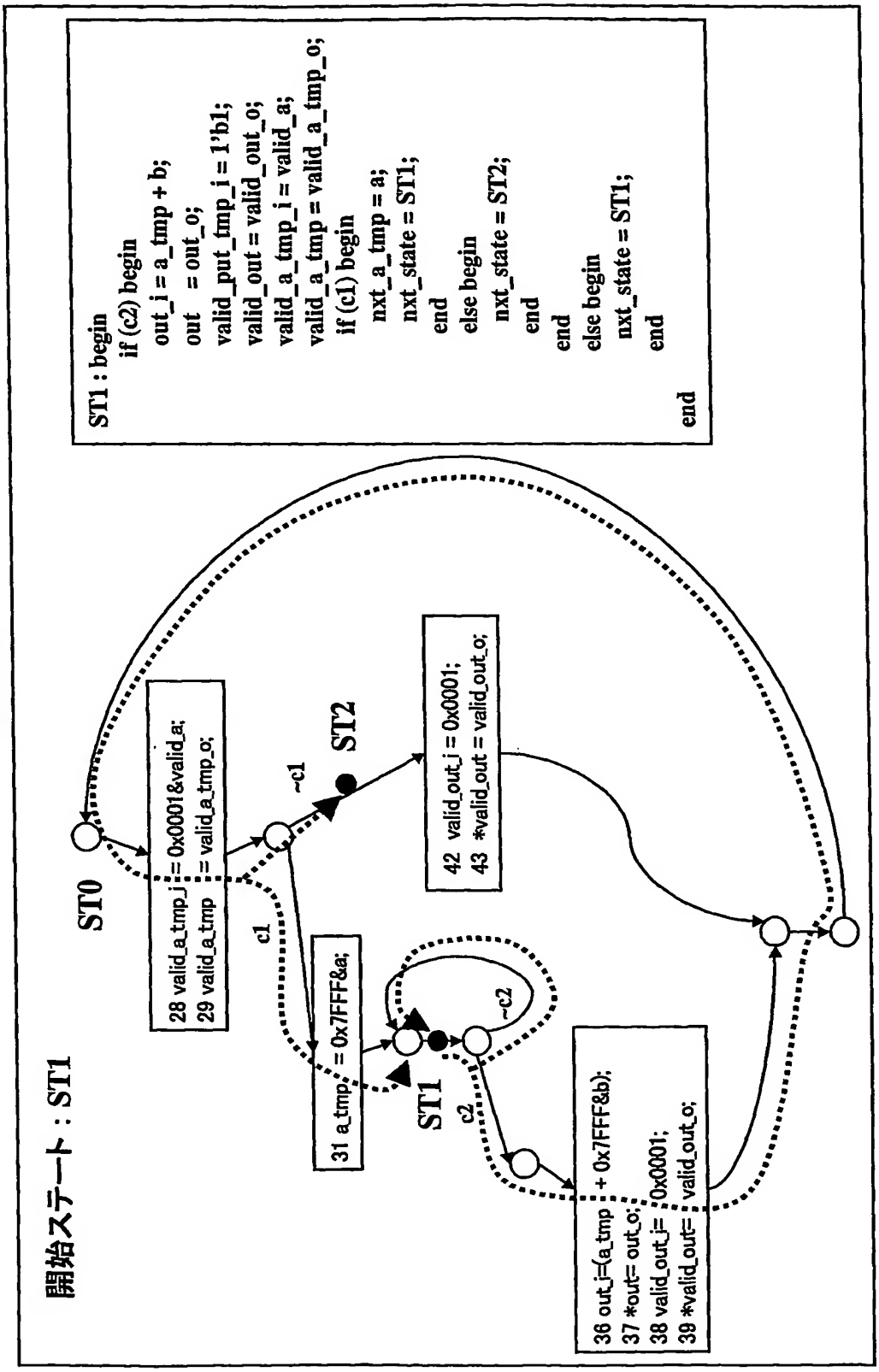
第５４図


開始ステート：ST0
ST0
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
~c1
ST2
31a_tmp = 0x7FFF&a;
ST1
c2
~c2
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
36 out_i = (a_tmp + 0x7FFF&b);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
各ステートからDFSでコードを生成する
特に、ステート変数は、
nxt_state = ST0;
等の形式でコード生成を行う
ST0 : begin
valid_a_tmp_i = valid_a;
valid_a_tmp =
valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_state = ST2;
end
end
ST1への分岐
ST2への分岐

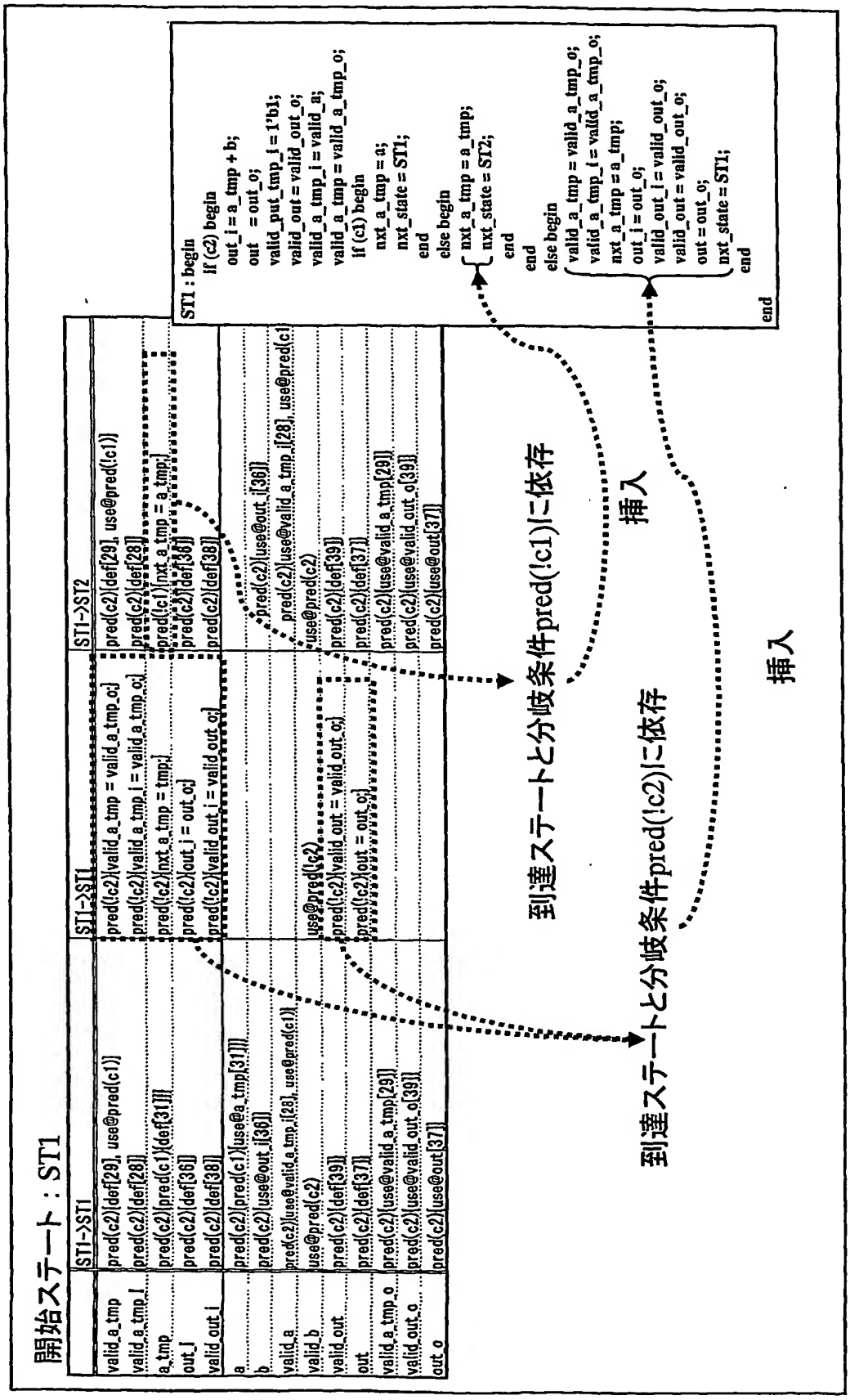
第５５図


開始ステート：ST0
ST0->ST1
ST0->ST2
valid_a_tmp
valid_a_tmp_I
a_tmp
out_I
valid_out_i
a
b
valid_a
valid_b
valid_out
out
valid_a_tmp_o
valid_out_o
out_o
def[29] use@pred(c1)
def[28]
pred(c1)|def[31]
out_i = out_o;
valid_out_i = valid_out_o;
pred(c1)|use@a_tmp[31]
use@valid_a_tmp_i[28]
use@pred(c1)
valid_out = valid_out_o;
out = out_o;
use@valid_a_tmp[29]
def[29] use@pred(!c1)
def[28]
pred(!c1)|nxt_a_tmp=tmp;
out_i = out_o;
valid_out_i = valid_out_o;
use@valid_a_tmp_i[28] use@pred(!c1)
開始ステートにのみ依存
挿入
ST0 : begin
out_i = out_o;
valid_out_i = valid_out_o;
valid_out = valid_out_o;
out = out_o;
valid_a_tmp_i = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_a_tmp = a_tmp;
nxt_state = ST2;
end
end
nxt_付きの変数名を使用
到達ステートと分岐条件pred(!c1)に依存
挿入

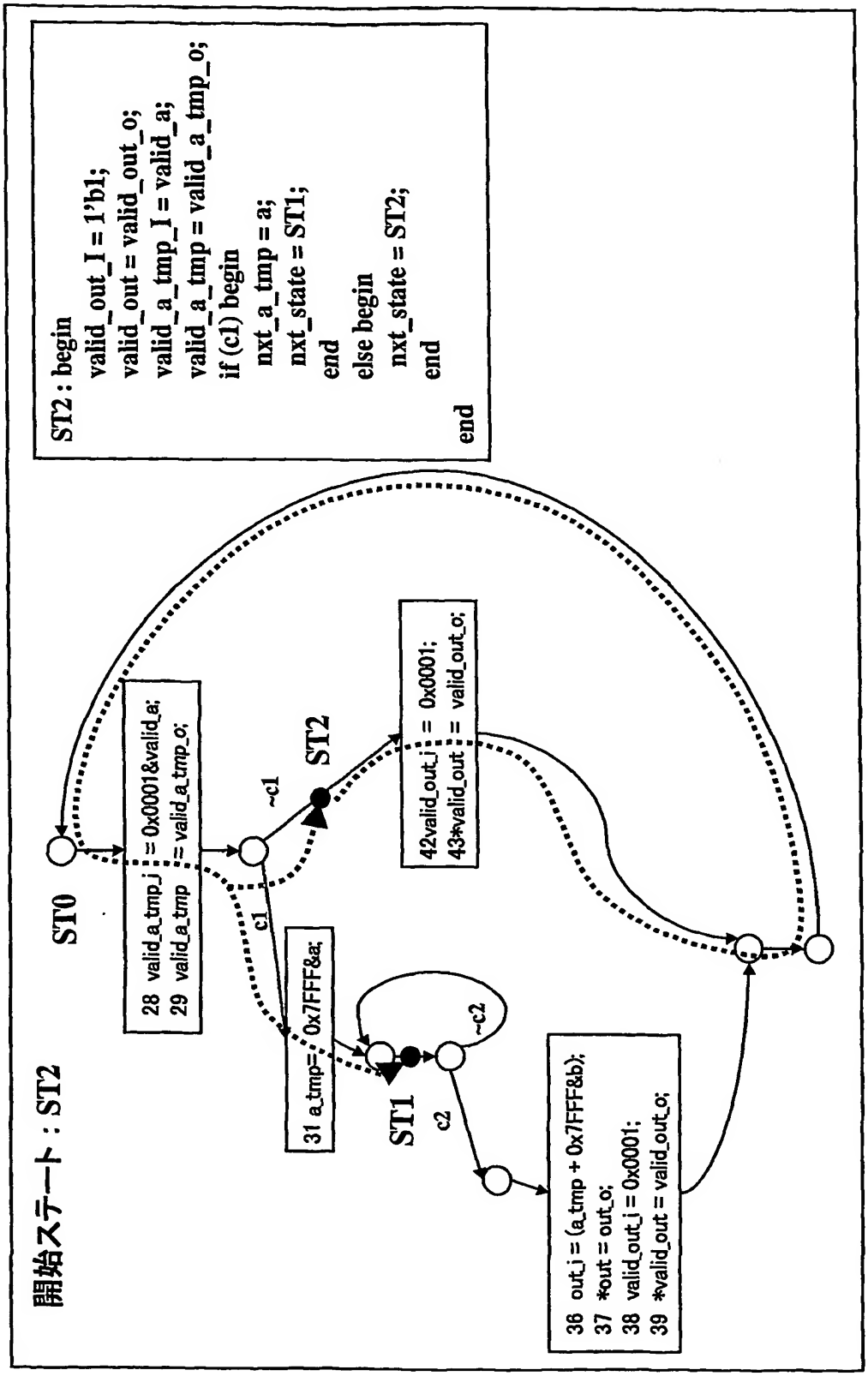
第56図


ST1 : begin
if (c2) begin
out_i = a_tmp + b;
out = out_o;
valid_put_tmp_i = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_i = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_state = ST2;
end
end
else begin
nxt_state = ST1;
end
end
開始ステート：ST1
ST0
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
~c1
ST2
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp_i = 0x7FFF&a;
ST1
c2
~c2
36 out_i=(a_tmp + 0x7FFF&b);
37 *out= out_o;
38 valid_out_i= 0x0001;
39 *valid_out= valid_out_o;

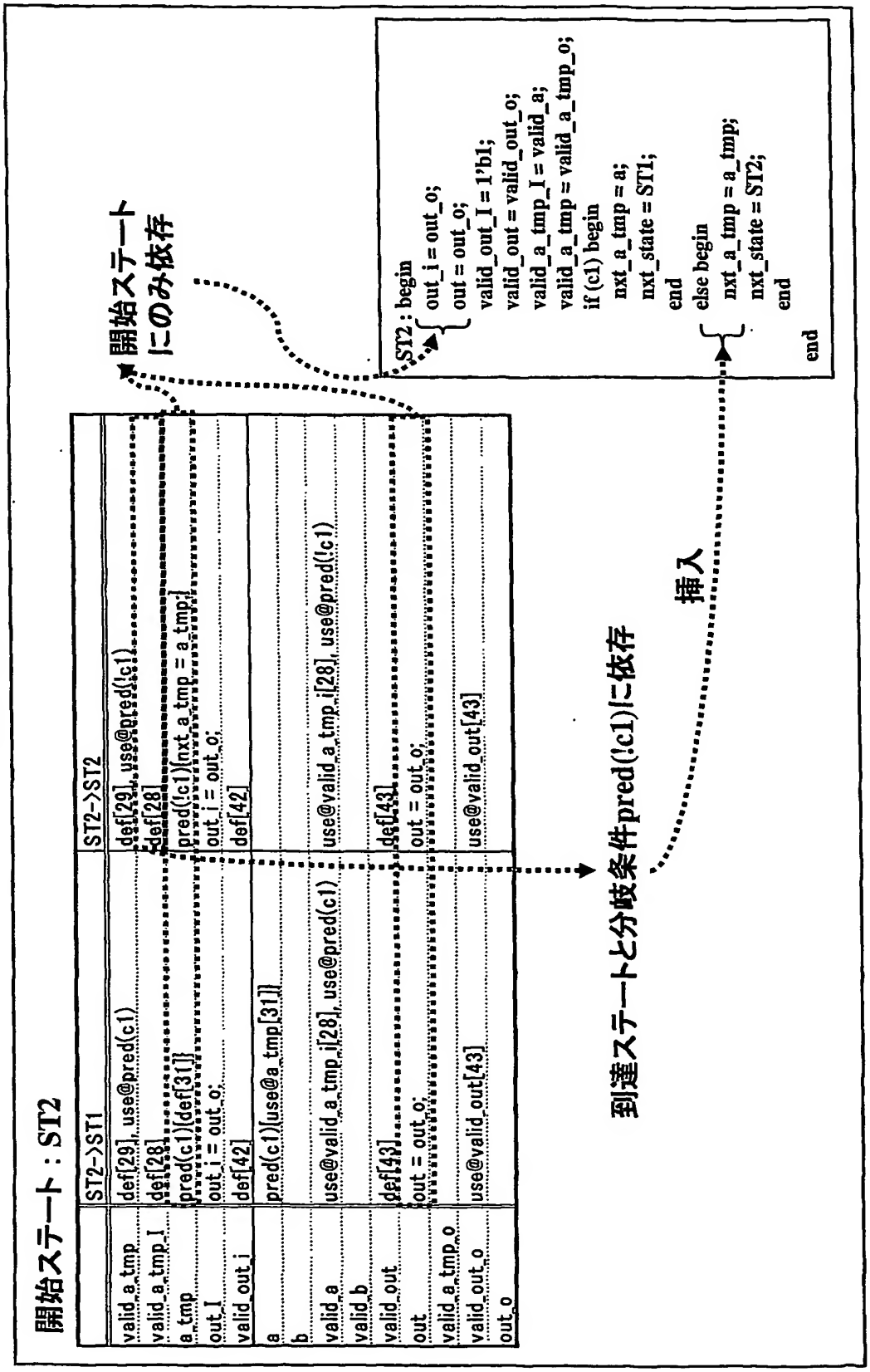
第５７図


開始ステート：ST1
ST1->ST1
ST1->ST2
valid_a_tmp
valid_a_tmp_i
a_tmp
out_i
valid_out_i
a
b
valid_a
valid_b
valid_out
out
valid_a_tmp_o
valid_out_o
out_o
pred(c2)[def[29], use@pred(!c1)]
pred(!c2)[valid_a_tmp = valid_a_tmp_o;]
pred(!c2)[def[28]]
pred(!c2)[valid_a_tmp_i = valid_a_tmp_o;]
pred(!c2)[pred(!c1)[def[31]]]
pred(!c2)[nxt_a_tmp = tmp;]
pred(!c2)[out_i = out_o;]
pred(!c2)[valid_out_i = valid_out_o;]
pred(c2)[def[38]]
pred(!c2)[use@a_tmp[31]]]
pred(c2)[use@out_i[36]]
pred(c2)[use@valid_a_tmp_i[28], use@pred(c1)]
use@pred(!c2)
pred(!c2)[def[39]]
pred(!c2)[def[37]]
pred(c2)[use@valid_a_tmp[29]]
pred(c2)[use@valid_out_o[39]]
pred(c2)[use@out[37]]
pred(c2)[def[29], use@pred(!c1)]
pred(c2)[def[28]]
pred(!c1)[nxt_a_tmp = tmp;]
pred(c2)[def[36]]
pred(c2)[def[38]]
pred(c2)[use@out_i[36]]
pred(c2)[use@valid_a_tmp_i[28], use@pred(c1)]
use@pred(!c2)
pred(!c2)[def[39]]
pred(!c2)[def[37]]
pred(c2)[use@valid_a_tmp[29]]
pred(c2)[use@valid_out_o[39]]
pred(c2)[use@out[37]]
ST1 : begin
If (c2) begin
out_i = a_tmp + b;
out = out_o;
valid_out_tmp_i = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_i = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_a_tmp = a_tmp;
nxt_state = ST2;
end
end
else begin
valid_a_tmp = valid_a_tmp_o;
valid_a_tmp_i = valid_a_tmp_o;
nxt_a_tmp = a_tmp;
out_i = out_o;
valid_out_i = valid_out_o;
valid_out = valid_out_o;
out = out_o;
nxt_state = ST1;
end
end
到達ステートと分岐条件pred(!c1)に依存
挿入
到達ステートと分岐条件pred(!c2)に依存
挿入

第58図


開始ステート：ST2
ST0
ST1
ST2
~c1
c1
c2
~c2
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
31 a_tmp=~0x7FFF&a;
42valid_out_i = 0x0001;
43*valid_out = valid_out_o;
36 out_i = (a_tmp + 0x7FFF&b);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
ST2 : begin
valid_out_I = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_I = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_state = ST2;
end
end

第５９図


開始ステート：ST2
ST2->ST1
ST2->ST2
valid_a_tmp
valid_a_tmp_I
a_tmp
out_I
valid_out_i
a
b
valid_a
valid_b
valid_out
out
valid_a_tmp_o
valid_out_o
out_o
def[29], use@pred(c1)
def[28]
pred(c1)[def[31]]
out_i = out_o;
def[42]
pred(c1)[use@a_tmp[31]]
use@valid_a_tmp_i[28], use@pred(c1)
def[43]
out = out_o;
use@valid_out[43]
def[29], use@pred(!c1)
def[28]
pred(!c1)[nxt_a_tmp = a_tmp;]
out_i = out_o;
def[42]
use@valid_a_tmp_i[28], use@pred(!c1)
def[43]
out = out_o;
use@valid_out[43]
開始ステート
にのみ依存
到達ステートと分岐条件pred(!c1)に依存
挿入
ST2 : begin
out_i = out_o;
out = out_o;
valid_out_I = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_I = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_a_tmp = a_tmp;
nxt_state = ST2;
end
end

第６０図

```
1  module PipeLine(clk, reset_n,
2                  valid_a, valid_b, a, b,
3                  out, valid_out);
   // System clock and reset
4  input clk;
5  input reset_n;
   // PipeLine input signals
6  input valid_a;
7  input valid_b;
8  input [14:0] a;
9  input [14:0] b;
   // PipeLine output signals
10 output valid_out;
11 reg valid_out;
12 output [15:0] out;
13 reg [15:0] out;
   // PipeLine internal signals
14 reg valid_a_tmp;
15 reg valid_a_tmp_i;
16 reg valid_a_tmp_o;
17 reg [14:0] a_tmp;
18 reg [14:0] nxt_a_tmp;
19 reg valid_out_i;
20 reg valid_out_o;
21 reg [15:0] out_i;
22 reg [15:0] out_o;
   // State registers
23 reg [1:0] state, nxt_state;
24 parameter ST0=2'b00,
25           ST1=2'b01,
26           ST2=2'b10;
   // Blanch conditions
27 wire c1;
28 wire c2;
29 assign c1 = !valid_a_tmp&&valid_a;
30 assign c2 = valid_b;
```

第６１図

```
    // Regsiter assignment statement
31  always @ (posedge clk or negedge reset_n) begin
32    if (!reset_n) begin
33      valid_a_tmp_o <= 1'b0;
34      out_o       <= 17'b00000000000000000;
35    end
36    else begin
37      valid_a_tmp_o <= valid_a_tmp_i;
38      out_o       <= out_i;
39    end
40  end
    // State registers and temporal registers
41  always @ (posedge clk or negedge reset_n) begin
42    if (!reset_n) begin
43      state       <= ST0;
44      a_tmp       <= 16'b0;
45    end
46    else begin
47      state       <= nxt_state;
48      a_tmp       <= nxt_a_tmp;
49    end
50  end
```

```
    // Mealy finite state machine
51  always @ (state or c1 or c2 or
52              valid_a_tmp_i or valid_a_tmp_o or
53              valid_a_tmp or a_tmp or
54              valid_out_i or valid_out_o or
55              out_i or out_o) begin
56    case(state[1:0])
57    ST0 : begin
58          valid_a_tmp_i  = valid_a;
59          valid_a_tmp = valid_a_tmp_o;
60          valid_out_i = valid_out_o;
61          valid_out     = valid_out_o;
62          out_i         = out_o;
63          out           = out_o;
64          if (c1) begin
65            nxt_a_tmp = a;
66            nxt_state = ST1;
67          end
68          else begin
69            nxt_a_tmp = a_tmp;
70            nxt_state = ST2;
71          end
72    end
```

第６２図

```
72    ST1 : begin
73              if (c2) begin
74                out_j          = a_tmp + b;
75                out            = out_o;
76               valid_out_j = 1'b1;
77               valid_out      = valid_out_o;
78                valid_a_tmp_j  = valid_a;
79                valid_a_tmp = valid_a_tmp_o;
80                if (c1) begin
81                  nxt_a_tmp = a;
82                  nxt_state = ST1;
83                end
84                else begin
85                  nxt_a_tmp = a_tmp;
86                  nxt_state = ST2;
87                end
88              end
89              else begin
90                nxt_state      = ST1;
91                valid_a_tmp_j  = valid_a_tmp_o;
92                valid_a_tmp = valid_a_tmp_o;
93                nxt_a_tmp      = a_tmp;
95               valid_out_j = valid_out_o;
96              valid_out      = valid_out_o;
97              out_j          = out_o;
98              out            = out_o;
99              end
100   end
```

```
101   ST2 : begin
102             valid_a_tmp_j  = valid_a;
103             valid_a_tmp = valid_a_tmp_o;
104            valid_out_j = 1'b0;
105            valid_out      = valid_out_o;
106           out_j          = out_o;
107           out            = out_o;
108            if (c1) begin
109              nxt_a_tmp = a;
110              nxt_state = ST1;
111            end
112            else begin
113              nxt_a_tmp = a_tmp;
114              nxt_state = ST2;
115            end
116   end
117   default : begin
118    nxt_state      = ST0;
119    valid_a_tmp_j  = valid_a_tmp_o;
120    valid_a_tmp = 1'b0;
121    nxt_a_tmp      = 15'b0;
122    valid_out_j = valid_out_o;
123    valid_out      = 1'b0;
124    out_j          = out_o;
125    out            = out_o;
126   end
127   endcase
128 end
129endmodule
```

INTERNATIONAL SEARCH REPORT

International application No.
PCT/JP03/12839

A. CLASSIFICATION OF SUBJECT MATTER
Int.Cl[7] G06F17/50

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
Int.Cl[7] G06F17/50

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X Y	Kazutoshi WAKABAYASHI et al., "Densoyo LSI o Dosa Gosei de Kaihatsu, Kino Sekkei no Kikan ga 1/10 ni Tanshuku", Nikkei Electronics, Nikkei Business Publications, Inc., 12 February, 1996 (12.02.96), No.655, pages 147 to 169	1,4-6 1-14
Y	JP 2002-49652 A (Hiroshi YASUDA), 15 February, 2002 (15.02.02), Claims 1 to 5 (Family: none)	1-14
A	KUROKAWA, H. et al., "C++ Based System Simulator for Pre-Verification of System-on-a-Chip Devices", NEC Research & Development, 07 December, 2000 (07.12.00), Vol.41, No.3, pages 258 to 263	10,12-13

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier document but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
29 October, 2003 (29.10.03)	18 November, 2003 (18.11.03)
Name and mailing address of the ISA/ Japanese Patent Office Facsimile No.	Authorized officer Telephone No.

Form PCT/ISA/210 (second sheet) (July 1998)

国際調査報告

国際出願番号　ＰＣＴ／ＪＰ０３／１２８３９

A.　発明の属する分野の分類（国際特許分類（ＩＰＣ））

Ｉｎｔ．Ｃｌ．' Ｇ０６Ｆ１７／５０

B.　調査を行った分野

調査を行った最小限資料（国際特許分類（ＩＰＣ））

Ｉｎｔ．Ｃｌ．' Ｇ０６Ｆ１７／５０

最小限資料以外の資料で調査を行った分野に含まれるもの

国際調査で使用した電子データベース（データベースの名称、調査に使用した用語）

C.　関連すると認められる文献

引用文献のカテゴリー＊	引用文献名　及び一部の箇所が関連するときは、その関連する箇所の表示	関連する請求の範囲の番号
X Y	若林一敏、外７名、“伝送用ＬＳＩを動作合成で開発，機能設計の期間が１／１０に短縮”、日経エレクトロニクス、日経ＢＰ社、1996.02.12、No.655、P.147-169	1,4-6 1-14
Y	JP 2002-49652 A(安田博)2002.02.15、請求項１－５（ファミリーなし）	1-14
A	Kurokawa, H. et al. “C++ Based System Simulator for Pre-Verification of System-on-a-Chip Devices”、NEC Research & Development、2000.12.07、Vol.41、No.3、p.258-263	10,12-13

☐ Ｃ欄の続きにも文献が列挙されている。　☐ パテントファミリーに関する別紙を参照。

＊　引用文献のカテゴリー
「Ａ」特に関連のある文献ではなく、一般的技術水準を示すもの
「Ｅ」国際出願日前の出願または特許であるが、国際出願日以後に公表されたもの
「Ｌ」優先権主張に疑義を提起する文献又は他の文献の発行日若しくは他の特別な理由を確立するために引用する文献（理由を付す）
「Ｏ」口頭による開示、使用、展示等に言及する文献
「Ｐ」国際出願日前で、かつ優先権の主張の基礎となる出願の日の後に公表された文献
「Ｔ」国際出願日又は優先日後に公表された文献であって出願と矛盾するものではなく、発明の原理又は理論の理解のために引用するもの
「Ｘ」特に関連のある文献であって、当該文献のみで発明の新規性又は進歩性がないと考えられるもの
「Ｙ」特に関連のある文献であって、当該文献と他の１以上の文献との、当業者にとって自明である組合せによって進歩性がないと考えられるもの
「＆」同一パテントファミリー文献

国際調査を完了した日	国際調査報告の発送日
２９．１０．０３	18.11.03

国際調査機関の名称及びあて先	特許庁審査官（権限のある職員）		
日本国特許庁（ＩＳＡ／ＪＰ） 郵便番号１００－８９１５ 東京都千代田区霞が関三丁目４番３号	早川　学	５Ｈ	９６５２
	電話番号　０３－３５８１－１１０１　内線　３５３１		

様式ＰＣＴ／ＩＳＡ／２１０（第２ページ）（１９９８年７月）